b
UNIVERSITÄT
BERN

Smelly APIs in Android ICC

Analysis of source code and relevant metadata

Bachelor Thesis

Astrid Ytrehorn
from
Volda, Norway

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Summer 2018

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

The Android ecosystem allows development of apps with relative ease through the extensive Android API. When developing the apps, security issues are often overlooked by the developers. This thesis is based on a previous work which identified 12 such Inter Component Communication (ICC) security smells that can lead to numerous security breaches in the system. A static code analysis tool based on Android Lint was developed to identify them. To further understand why some of these smells are so prominent, this thesis evaluated their appearances based on several aspects. First the influence of developers in the projects was examined. The association of developers to different apps was cross-referenced with the occurrence of smells per project and we found that for most smells the developers have a tendency to make the mistake over more than one project. We also examined how updates affect smells. The updates rarely brought a change in smells and if they did they tended to have a negative impact. We performed a manual analysis of 100 apps with the most smells. The lint-based tool was found to have a good and correct detection rate. In the next study we examined if the smells that went unreported by the tool were correctly labeled as such and the reason for not them not being detected. In most cases this was due to the relevant Android API not being used. Finally, we did a study on the location of smells in the code base. We expanded the existing linting tool to include more metadata and analyzed all the apps once more. The different smell categories tended to have a varying degree of displacement of individual smells in the code base. The average number of distinct locations grew in the order of Java package, containing class and surrounding method for most of the smells. This thesis aims to help spread awareness abut ICC security smells and thereby fundamentally reduce the attack surface in Android.

Contents

1 Introduction

Smartphones and tablets provide powerful features once offered only by computers. However, the risk of security vulnerabilities on these devices is tremendous; smartphones are increasingly used for security-sensitive services like e-commerce, e-banking, and personal healthcare, which make these multi-purpose devices an irresistible target of attack for criminals.

A recent survey on the StackOverflow website shows that about 65% of mobile developers work with Android.[1] This platform has captured over 80% of the smartphone market,[2] and just its official app store contains more than 2.8 million apps. As a result, a security mistake in an in-house app may jeopardize the security and privacy of billions of users.

The security of smartphones has been studied from various perspectives such as the device manufacturer [26], its platform [29], and end users [10]. Numerous security APIs, protocols, guidelines, and tools have been proposed. Nevertheless security concerns are often overridden by other concerns [3]. Many developers undermine their significant role in providing security [27]. As a result, security issues in mobile apps continue to proliferate unabated.[3]

Given this situation, a previous work identified 28 security code smells *i.e.,* symptoms in the code that signal the prospect of security vulnerabilities [9]. The prevalence of ten of such smells was studied, and the researchers realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different

[1] `http://insights.stackoverflow.com/survey/2017`
[2] `http://www.gartner.com`
[3] `http://www.cvedetails.com`

security smells, and such smells are in fact good indicators of security vulnerabilities.

Android Inter-Component Communication (ICC) is complex, largely unconstrained, and hard for developers to understand, and it is consequently a common source of security vulnerabilities in Android apps. In a previous work, twelve security code smells pertinent to ICC vulnerabilities were identified and are investigated further in this thesis.

In a previous work a software tool based on Android Lint was developed to statically analyze the source code of Android apps. The correctness of this tool is analyzed in this thesis. Furthermore, metadata of a large number of Android app projects was gathered and cross-referenced with the appearing security issues. The following research questions are addressed:

- **RQ**$_1$: *How does team size play a role for security issues in Android apps?* There was a review to find a correlation between the number of contributors for an app and the number of security smells found. This was done by analyzing metadata about contributors from each project's Github repository (where available). We found that larger teams tend to generally produce more security issues.

- **RQ**$_2$: *How, if at all, do the number of security smells change in subsequent app releases, and why?* After a manual evaluation of the number of smells for subsequent app releases, we found that when the security smells change for the worse, it is usually because new ICC-related features are introduced with corresponding security issues. The existing smells are rarely fixed with version updates.

- **RQ**$_3$: *How does project age and activity influence security issues?* We found that older projects tend to have fewer security smells than newer ones. This was evaluated by again looking at a project's Github metadata, namely the commits done by all contributors. Similarily, we found that apps that have more frequent developer activity (e.g. updates), tend to have more smells.

- **RQ**$_4$: *How reliable is the linting tool used?* A manual analysis of 100 apps with the most smells showed that the linting tool is reliable. The smells found by it corresponded well with the findings of the developers doing the manual analysis.

- **RQ**$_5$: *Why are some security smells not always detected?* In each case; is it because the API is not being used, because the smell is mitigated or is it because the tool failed to detect it? Using regular expressions we found that when smells are not detected, it is usually because the API is not being used. There were a few cases in which the tool failed to detect them, but that was due to complex code semantics.

- **RQ**$_6$: *How are ICC security smells placed in the apps?* Are they gathered in the same packages, classes and methods or are they scattered across the app. To

find the location of smells in the app, we expanded the existing lint tool to give more information about smell placement in the app. The magnitude of spreading across the code base increases in the following order: package, class and method. This was expected behaviour, however there were some cases where the spread in classes was larger than methods. Some projects showed significant distribution of smells in the code base due to project size.

To summarize, this work is part of an initial effort to spread awareness about the impact of programming choices in making apps secure, and to fundamentally reduce the attack surface in Android.

The remainder of this thesis is organized as follows. We provide the necessary background about the Android OS and ICC risks from which Android apps suffer and we introduce ICC-related security code smells in chapter 2, followed by an empirical study in chapter 3. We make conclusions in chapter 4, while chapter 5 discusses threats of validity. We provide a brief overview of the related work in chapter 6. Finally chapter 7 gives a tutorial for developers on how to avoid the security smells mentioned in chapter 2.

2 ICC Security code smells

This bachelor thesis builds on a study that identified twelve ICC security code smells [7]. These are presented in this chapter.

2.1 Background

First, this section covers the necessary background in the Android platform, and briefly presents common security threats in the context of ICC scenarios.

2.1.1 Android Architecture

Android is the most popular operating system (OS) for smartphones and other types of mobile devices. It provides a rich set of APIs for app developers to access common features on mobile devices.

An Android app consists of an .apk file containing the compiled bytecode, any needed data, and resource files. The Android platform assigns a unique user identifier (UID) to each app at installation time, and runs it in a unique process within a sandbox so that every app runs in isolation from other apps. Moreover, access to sensitive APIs is protected by a set of permissions that the user can grant to an app. In general, these permissions are text strings that correlate to a specific access grant, *e.g.,* `android.permission.CAMERA` for camera access.

Four types of components can exist in an app: activities, services, broadcast receivers, and content providers. In a nutshell:

- *Activities* build the user interface of an app, and allow users to interact with the app.
- *Services* run operations in the background, without a user interface.
- *Broadcast receivers* receive system-wide "intents", *i.e.,* descriptions of operations to be performed, sent to multiple apps. Broadcast receivers act in the background, and often relay messages to activities or services.
- *Content providers* manage access to a repository of persistent data that could be used internally or shared between apps.

The OS and its apps, as well as components within the same or across multiple apps, communicate with each other via ICC APIs. These APIs take an *intent object* as a parameter. An intent is either *explicit* or *implicit*. In an explicit intent, the source component declares to which target component (*i.e.,* Class or ComponentName instances) the intent is sent, whereas in an implicit intent, the source component only specifies a general action to be performed (*i.e.,* represented by a text string), and the target component that will receive the intent is determined at runtime. Intents can optionally carry additional data also called *bundles*. Components declare their ability to receive implicit intents using "intent filters", which allow developers to specify the kinds of actions a component supports. If an intent matches any intent filter, it can be delivered to that component.

2.1.2 ICC Threats

ICC not only significantly contributes to the development of collaborative apps, but it also poses a common attack surface. The ICC-related attacks that threaten Android apps are:

- Denial of Service. Unchecked exceptions that are not caught will usually cause an app to crash. The risk is that a malicious app may exploit such programming errors, and perform an inter-process denial-of-service attack to drive the victim app into an unavailable state.
- Intent Spoofing. In this scenario a malicious app sends forged intents to mislead a receiver app that would otherwise not expect intents from that app.
- Intent Hijacking. This threat is similar to a man-in-the-middle attack where a malicious app, registered to receive intents, intercepts implicit intents before they reach the intended recipient, and without the knowledge of the intent's sender and receiver.

Two major consequences of the ICC attacks are as follows:

- Privilege Escalation. The security model in Android does not by default prevent an app with fewer permissions (low privilege) from accessing components of another app with more permissions (high privilege). Therefore, a caller can escalate its permissions via other apps, and indirectly perform unauthorized actions through the callee.
- Data Leak. A data leak occurs when private data leaves an app and is disclosed to an unauthorized recipient.

2.2 General Overview

This section gives a general overview of each individual smell and how it affects security if abused.

- **SM01: Persisted Dynamic Permission** — Android uses Uniform Resource Identifiers (URI) to provide access to protected resources at runtime. If the developer forgets to revoke such a permission, the access persists and could lead to sensitive data remaining exposed.
- **SM02: Custom Scheme Channel** — By using a *custom scheme*, a developer can register their app for custom URIs to make it responsive to URIs tailored to the app. However, any app can register any custom schemes and access URIs containing e.g. access tokens or credentials.
- **SM03: Incorrect Protection Level** — To access sensitive resources, apps must request permission. Additionally, a developer can set *custom permissions* to restrict access to specific features. Depending on the permission the user might be prompted to grant access or not. If a permission is declared in an app and is given the incorrect protection level, apps might still be able to use a protected feature.
- **SM04: Unauthorized Intent** — Intents are used for one-way requests (eg. sending text messages). Receivers can set custom permissions without which clients are not allowed to communicate with them. Any app can send an unprotected intent or can register to receive unprotected ones. Implicit intents (that do not specify a target but a general action) are unprotected and could lead to a privilege escalation to privileged targets. Similarily, malicious apps receiving unprotected intents may leak or manipulate their data and relay them.
- **SM05: Sticky Broadcast** — Normally a broadcast terminates after it reaches the receivers it was intended for. *Sticky broadcasts* persist so that they can notify other

apps if they need that specific information. This is for example used to broadcast battery life to all necessary apps. This can however be abused because an app can register to broadcasts and tamper with them.

- **SM06: Slack WebViewClient** — *WebView* is used to allow web browsing within an app. By default, the activity manager is asked to choose a handler for the URL. If a *WebViewCient* is provided to the *WebView* then the host application handles the URL. The problem is that the default client does not restrict access to any page and could be used to access malicious websites.

- **SM07: Broken Service Permission** — There are two ways to start a service. Either onBind or onStartCommand. The latter allows a service to persist in the background, even after the client disconnects. Apps that use Android IPC to start a service may not possess the same permission as the service provider itself. Consequently, if a service is exposed it can be abused. Should the callee be in possession of the required permissions, the caller will also gain access to the service. This could lead to privilege escalation.

- **SM08: Insecure Path Permission** — Data can be shared with other apps and besides regular permissions that apply, a developer can set path-specific permissions to allow a more granular control. Normally the path-permission check differentiates between double slashes and single slashes in the path. If there is a mismatch only the permission on the whole content provider is consider. However, the *UriMatcher* provided by Android considers double- and single-slash paths to be identical and will forward such requests, which can lead to unwanted access to protected resources.

- **SM09: Broken Path Permission Precedence** — In a content provider, permissions that are more granular should take precedence over larger-scope ones. However due to a bug in a recent release of the Android framework a path permission does not take precedence over permission on the provider as a whole. Therefore content providers may grant access to apps where access was not intended.

- **SM10: Unprotected Broadcast Receiver** — Static broadcast receivers are set in the Android manifest, and start even if an app is not currently active. Dynamic broadcast receivers are registered at run time and execute only when the app is active. Any app can register to receive a broadcast, which exposes it to other apps that are able to send that broadcast. If no permission check is done, the receiver might respond to a spoofed broadcast which could lead to data leaks.

- **SM11: Implicit Pending Intent** — *Pending intents* are set to be executed in the future on behalf of an app. Any app can receive implicit pending intents, as the implicit intents do not target specific components and just contain an action they

wish to perform. The intercepted intent can be abused to send arbitrary intents on behalf of the initial app. This can lead to apps tampering with the original intents data and performing actions with the permissions of the original sender. This could be used for spoofing attacks.

- **SM12: Common Task Affinity** — Tasks are a collection of activities that a user interacts with when performing a certain job. Task affinity defines which activity a task wants to belong to. By default the activities in an app prefer to be in the same task and have the same affinity. If apps use the same task affinity, their activities can overlap, which can be abused to hijack another app's activity and use the activity for spoofing.

2.3 Detailed Explanation

This section is quoted from Gadient *et al.* [7] and covers each smell in greater detail for a better understanding for the reader.

For each smell these things are reported; the security *issue* at stake, the potential security *consequences* for users, the *symptom* in the code (*i.e.,* the code smell), the *detection* strategy that has been implemented by the tool for identifying the code smell, any *limitations* of the detection strategy, and a recommended *mitigation* strategy of the issue, principally for developers.

- **SM01: Persisted Dynamic Permission** Android provides access to protected resources through a Uniform Resource Identifier (URI) to be granted at runtime.
  Issue: Such dynamic access is intended to be temporary, but if the developer forgets to revoke a permission, the access grant becomes more durable than intended.
  Consequently, the recipient of the granted access obtains long-term access to potentially sensitive data.
  Symptom: `Context.grantUriPermission()` is present in the code without a corresponding `Context.revokeUriPermission()` call.
  Detection: The smell is reported when a permission being dynamically granted without any revocations in the app is detected.
  Limitation: The implementation does not match a specific grant permission to its corresponding revocation. It may therefore fail to detect a missing revocation if another revocation is present somewhere in the code.
  Mitigation: Developers have to ensure that granted permissions are revoked when they are no longer needed. They can also attach sensitive data to the intent instead of providing its URI.

- **SM02: Custom Scheme Channel** A *custom scheme* allows a developer to register an app for custom URIs, *e.g.,* URIs beginning with `myapp://`, throughout the

operating system once the app is installed. For example, the app could register an activity to respond to the URI via an intent filter in the manifest. Therefore, users can access the associated activity by opening specific hyperlinks in a wide set of apps.
Issue: Any app is potentially able to register and handle any custom schemes used by other apps.
Consequently, malicious apps could access URIs containing access tokens or credentials, without any prospect for the caller to identify these leaks [23].
Symptom: If an app provides custom schemes, then a scheme handler exists in the manifest file or in the Android code. If the app calls a custom scheme, there exists an intent containing a URI referring to a custom scheme.
Detection: The `android:scheme` attribute exists in the `intent-filter` node of the manifest file, or `IntentFilter.addDataScheme()` exists in the source code.
Limitation: Only the symptoms related to receiving custom schemes are checked.
Mitigation: Never send sensitive data *e.g.,* access tokens via such URIs. Instead of custom schemes use system schemes that offer restrictions on the intended recipients. The Android OS could maintain a verified list of apps and the schemes that are matched when there is such call.

- **SM03: Incorrect Protection Level** Android apps must request permission to access sensitive resources. In addition, custom permissions may be introduced by developers to limit the scope of access to specific features that they provide based on the protection level given to other apps. Depending on the feature, the system might grant the permission automatically without notifying the user *i.e.,* signature level, or after the user approval during the app installation, *i.e.,* normal level, or may prompt the user to approve the permission at runtime, if the protection is at dangerous level.
  Issue: An app declaring a new permission may neglect the selection of the right protection level, *i.e.,* a level whose protection is appropriate with respect to the sensitivity of resources [16].
  Consequently, apps with inappropriate permissions can still use a protected feature.
  Symptom: Custom permissions are missing the right `android:protectionLevel` attribute in the manifest file.
  Detection: Missing protection level declarations for custom permissions are reported.
  Limitation: It cannot be determined if the level specified for a protection level is in fact right.
  Mitigation: Developers should protect sensitive features with dangerous or signature protection levels.

- **SM04: Unauthorized Intent** Intents are popular as one way requests, *e.g.,* sending a mail, or requests with return values, *e.g.,* when requesting an image file from a photo library. Intent receivers can demand custom permissions that clients have to obtain before they are allowed to communicate. These intents and receivers are "protected".
  Issue: Any app can send an unprotected intent without having the appropriate permission, or it can register itself to receive unprotected intents.
  Consequently, apps could escalate their privileges by sending unprotected intents to privileged targets, *e.g.,* apps that provide elevated features such as camera access. Also, malicious apps registered to receive implicit unprotected intents may relay intents while leaking or manipulating their data [5].
  Symptom: The existence of an unprotected implicit intent. For intents requesting a return value, the lack of check for whether the sender has appropriate permissions to initiate an intent.
  Detection: The existence of several methods on the `Context` class for initiating an unprotected implicit intent like `startActivity`, `sendBroadcast`, `sendOrderedBroadcast`, `sendBroadcastAsUser`, and `sendOrderedBroadcastAsUser`.
  Limitation: It is not verified, for a given intent requesting a return value, if the sender enforces permission checks for the requested action.
  Mitigation: Use explicit intents to send sensitive data. When serving an intent, validate the input data from other components to ensure they are legitimate. Adding custom permissions to implicit intents may raise the level of protection by involving the user in the process.

- **SM05: Sticky Broadcast** A normal broadcast reaches the receivers it is intended for, then terminates. However, a "sticky" broadcast stays around so that it can immediately notify other apps if they need the same information.
  Issue: Any app can watch a broadcast, and particularly a sticky broadcast receiver can tamper with the broadcast [16].
  Consequently, a manipulated broadcast may mislead future recipients.
  Symptom: Broadcast calls that send a sticky broadcast appear in the code, and the related Android system permission exists in the manifest file.
  Detection: The existence of methods such as `sendStickyBroadcast`, `sendStickyBroadcastAsUser`, `sendStickyOrderedBroadcast`, `sendStickyOrderedBroadcastAsUser`, `removeStickyBroadcast`, and `removeStickyBroadcastAsUser` on the `Context` object in the code and the `android.permission.BROADCAST_STICKY` permission in the manifest file are checked.
  Limitation: No limitations are known.
  Mitigation: Prohibit sticky broadcasts. Use a non-sticky broadcast to report that

something has changed. Use another mechanism, *e.g.,* an explicit intent, for apps to retrieve the current value whenever desired.

- **SM06: Slack WebViewClient** A `WebView` is a component to facilitate web browsing within Android apps. By default, a `WebView` will ask the Activity Manager to choose the proper handler for the URL. If a `WebViewClient` is provided to the `WebView`, the host application handles the URL.
  Issue: The default implementation of a `WebViewClient` does not restrict access to any web page [16].
  Consequently, it can be pointed to a malicious website that entails diverse attacks like phishing, cross-site scripting, *etc.*
  Symptom: The `WebView` responsible for URL handling does not perform adequate input validation.
  Detection: The `WebView.setWebViewClient()` exists in the code but the `WebViewClient` instance does not apply any access restrictions in `WebView.shouldOverrideUrlLoading()`, *i.e.,* it returns `false` or calls `WebView.loadUrl()` right away. Also, a smell is reported if the implementation of `WebView.shouldInterceptRequest()` returns `null`.
  Limitation: It is inherently difficult to evaluate the quality of an existing input validation.
  Mitigation: Use a white list of trusted websites for validation, and benefit from external services, *e.g.,* SafetyNet API,[1] that provide information about the threat level of a website.

- **SM07: Broken Service Permission** Two different mechanisms exist to start a service: `onBind` and `onStartCommand`. Only the latter allows services to run indefinitely in the background, even when the client disconnects. An app that uses Android IPC to start a service may possess different permissions than the service provider itself.
  Issue: When the callee is in possession of the required permissions, the caller will also get access to the service.
  Consequently, a privilege escalation could occur [16].
  Symptom: The lack of appropriate permission checks to ensure that the caller has access right to the service.
  Detection: The smell is reported when the caller uses `startService`, and then the callee uses `checkCallingOrSelfPermission`, `enforceCallingOrSelfPermission`, `checkCallingOrSelfUriPermission`, or `enforceCallingOrSelfUriPermission` to verify the permissions of the request. Calls on the `Context` object for permission check will then fail as the system mistakenly considers the callee's permission instead of the caller's.

[1] `https://developer.android.com/training/safetynet/safebrowsing.html`

Furthermore, reported are calls to `checkPermission`, `checkUriPermission`, `enforcePermission`, or `enforceUriPermission` methods on the `Context` object, when additional calls to `getCallingPid` or `getCallingUid` on the `Binder` object exist.
Limitation: It is currently not distinguished between checks executed in `Service.onBind` or `Service.onStartCommand`, and custom permission checks based on the user id with `getCallingUid` are not verified.
Mitigation: Verify the caller's permissions every time before performing a privileged operation on its behalf using `Context.checkCallingPermission()` or `Context.checkCallingUriPermission()` checks. If possible, do not implement `Service.onStartCommand` in order to prevent clients from starting, instead of binding to, a service. Ensure that appropriate permissions to access the service have been set in the manifest.

- **SM08: Insecure Path Permission** When sharing data with other apps, besides regular permissions that apply to the whole of a content provider, it is possible to set path-specific permissions that are more fine-grained.
Issue: The path-permission check in the manifest file differentiates between paths containing double slashes and paths with one slash. Hence, if there is a mismatch the permission only on the whole content provider is considered. However, the `UriMatcher` provided by the Android framework, which is recommended for URI comparison in the `query` method of a content provider, considers such paths to be identical, and will forward the request to the initially intended resource.
Consequently, access to presumably protected resources may be granted to unauthorized apps [16].
Symptom: A `UriMatcher.match()` is used for URI validation.
Detection: `path-permission` attributes in the manifest file, and `UriMatcher.match()` methods in the code are looked for.
Limitation: No limitations are known.
Mitigation: As long as the bug exists in the Android framework, use your own URI matcher.

- **SM09: Broken Path Permission Precedence** In a content provider, more fine-grained permissions should take precedence over those with larger scope.
Issue: A path permission does not take precedence over permission on the whole provider due to a bug that we identified in the `ContentProvider.enforceReadPermissionInner()` method in recent releases of the Android framework.[2]

[2]The bug can be found at line 574. The class is publicly available at `https://android.googlesource.com/platform/frameworks/base/+/oreo-r6-release/core/java/android/content/ContentProvider.java`

Consequently, content providers may mistakenly grant access to other apps.
Symptom: The content provider is protected by path-specific permissions.
Detection: WA `path-permission` in the definition of a content provider in the manifest file is looked for.
Limitation: No limitations are known.
Mitigation: As long as the bug exists in Android, instead of path permissions use a distinct content provider with a dedicated permission for each path.

- **SM10: Unprotected Broadcast Receiver** Static broadcast receivers are registered in the manifest file, and start even if an app is not currently running. Dynamic broadcast receivers are registered at run time in Android code, and execute only if the app is running.
  Issue: Any app can register itself to receive a broadcast, which exposes the app to any other app able to initiate the broadcast.
  Consequently, if there is no permission check, the receiver may respond to a spoofed intent yielding unintended behavior or data leaks [16].
  Symptom: The `Context.registerReceiver()` call without any argument for permission exists in the codestatic symptoms.
  Detection: Cases where the permission argument is missing or is `null` are reported.
  Limitation: We are not aware of the permissions' appropriateness.
  Mitigation: Register broadcast receivers with sound permissions.

- **SM11: Implicit Pending Intent** A `PendingIntent` is an intent that executes the specified action of an app in the future and on behalf of the app *i.e.,* with the identity and permissions of the app that sends the intent, regardless of whether the app is running or not.
  Issue: Any app can intercept an implicit pending intent [16] and use the pending intent's `send` method to submit arbitrary intents on behalf of the initial sender.
  Consequently, a malicious app can tamper with the intent's data and perform custom actions with the permissions of the originator. Relaying of pending intents could be used for intent spoofing attacks.
  Symptom: The initiation of an implicit `PendingIntent` in the code.
  Detection: A smell is reported if methods such as `getActivity`, `getBroadcast`, `getService`, and `getForegroundService` on the `PendingIntent` object are called, without specifying the target component *Limitation:* Arrays of pending intents are not yet supported in the analysis.
  Mitigation: Use explicit pending intents, as recommended by the official documentation.[3]

[3] `https://developer.android.com/reference/android/app/PendingIntent.html`

- **SM12: Common Task Affinity** A *task* is a collection of activities that users interact with when carrying out a certain job.[4] A task affinity, defined in the manifest file, can be set to an individual activity or at the application level.
  Issue: Apps with identical task affinities can overlap each others' activities, *e.g.,* to fade in a voice record button on top of the phone call activity.
  Consequently, malicious apps may hijack an app's activity paving the way for various kinds of spoofing attacks [19].
  Symptom: The task affinity is not empty.
  Detection: A smell is reported if the value of a task affinity is not empty.
  Limitation: No limitations are known.
  Mitigation: If a task affinity remains unused, it should always be set to an empty string on the application level. Otherwise set the task affinity only for specific activities that are safe to share with others. It is suggested that Android set the default value for a task affinity to empty. It may also add the possibility of setting a permission for a task affinity.

In summary, each security smell introduces a different set of vulnerabilities. A close relationship between the smells and the security risks was established with the purpose of providing accessible and actionable information to developers, as shown in Table 2.1.

Vulnerabilities	**Security code smells**
Denial of Service	SM01, SM02, SM03, SM04, SM06, SM07, SM10, SM12
Intent Spoofing	SM02, SM03, SM04, SM05, SM07, SM08, SM09, SM10, SM11
Intent Hijacking	SM02, SM03, SM04, SM05, SM10, SM11

Table 2.1: The relationship between vulnerabilities and security code smells

[4] `https://developer.android.com/guide/components/activities/tasks-and-back-stack.html`

3
Empirical study

In this section we introduce a dataset of more than 700 open-source Android projects that are mostly hosted on GitHub. This dataset was analyzed for the existance of security code smells using a static analysis tool (see section 3.2) and the results were checked for correctness by means of a manual analysis of a subset from all apps. To further gain insight into the relevance of project metadata for security smells, we developed a C# tool to gather more information like involved developers, last update and age for each project. Such information was analyzed to answer the first four RQs.

The results in section 3.2 suggest that although fewer than 10% of apps suffer from more than two ICC security smells, smaller teams tend to be more capable of consistently building software resistant to certain security code smells. With respect to app volatility, we discovered that updates rarely have any impact on ICC security, however, in case they have, they often correspond to new app features. On the other hand, we found that long-lived projects have more issues than recently created ones, except for apps that receive frequent updates, where the opposite is true. Moreover, the findings of Android Lint's security checks correlate to the detected security smells.

The manual investigation in section 3.3 confirms that the tool successfully finds many different ICC security code smells, and about 48% of them in fact represent vulnerabilities. The tool can consequently offer valuable support in security audits.

In section 3.4 we report on how and why certain smells remained undetected in the linting tool. We found that in most cases, it was due to the relevant API not being used. Certain edge cases proved to be false negatives, which was usually due to complex code semantics that avoided the detection pattern. Some of the smells were also correctly mitigated by the developers.

In section 3.5 we look into the remaining research question. To do this, we extended the linting tool to provide further metadata on smells about its location in the code. This data was then analyzed and we found that most apps have the smells concentrated in one place, except for SM04 and SM10 which tended to be more widespread.

3.1 Dataset

We collected all open-source apps from the F-Droid[1] repository as well as several other apps directly from GitHub[2]. From a total of 3 471 apps, 1 487 (42%) could be successfully built. In order to reduce the influence of individual projects, in case there existed more than one release of a project, only the latest one was considered. Finally, there were 732 apps (21%) in the dataset. The median project size in this dataset is about 1.2 MB, corresponding to 108 files.

Most of these projects could be found on GitHub, and therefore a tool that queries the GitHub WebAPI[3] was implemented to collect meta information such as the list of contributors, the creation date, and the date of the last commit to each project.

3.2 Batch Analysis

In this section, the results from three automated analyses are presented. To analyze security smells in the code, we used a static code analysis tool on the projects. A previous work details the development of this tool [7], which was based on Android Lint, from the official Android Studio IDE[4]. The first study is a metadata analysis on developer influence on security smells. The second analysis investigates the changes in smells as apps receive updates from their developers. The final study looks at the relation of project age and activity to security smells.

To get an overview of the number of individual smells over all projects and how many projects are affected by the different smells, we did an automated analysis of 729 projects. The first evaluation checked how many smells of each category are present in all projects. Figure 3.1 shows this.

The second evaluation checks, how many projects are affected by each smell category. Again this was done for all 729 projects (see Figure 3.2). So every column is out of a maximum of 1374.

It is clearly visible that SM04 and SM12 are the most common ones with SM10 being the third highest. SM12 was present in every single project as shown in Figure 3.2. This

[1] `https://f-droid.org/`
[2] `https://github.com/pcqpcq/open-source-android-apps`
[3] `https://developer.github.com/v3/`
[4] `https://sites.google.com/a/android.com/tools/tips/lint`

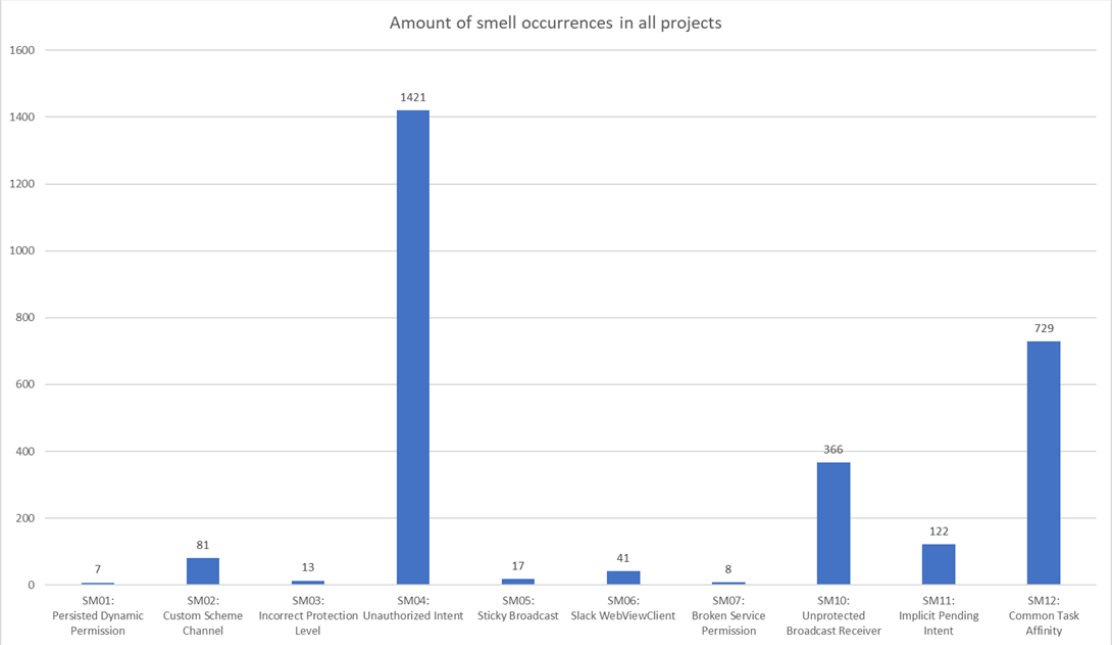


Figure 3.1: This figure shows how many smells of each category are present in 729 projects.

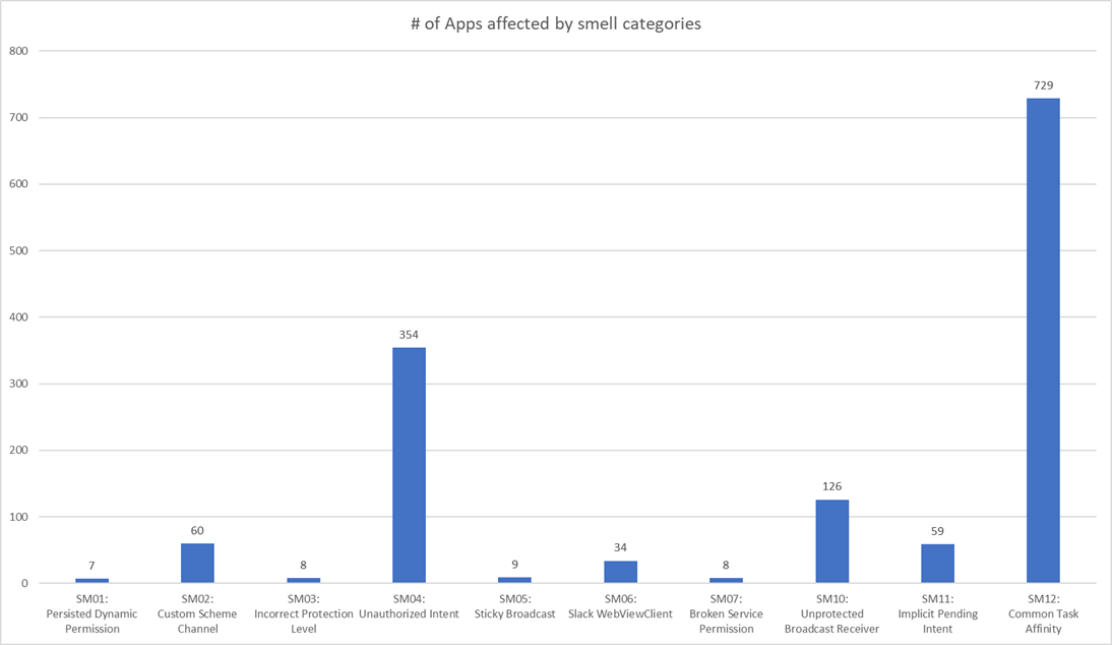


Figure 3.2: This figure shows how many projects are affected for each smell category in 729 projects.

means that developers are not aware of the security implications of using task affinity. Similarily, SM04 also appears in a large amount of projects. Developers do not seem to be aware of explicit intents being more secure than implicit ones. SM08 and SM09 were never detected in all projects, which is why they are missing in the figures.

3.2.1 Contributor Affiliation

To answer RQ_1 we implemented an F-Droid repo index parser[5] to extract any potential Github link for each app. Then we developed a C# tool to extract the information about contributors from the Github repositories using the official Github API (Octokit). The C# tool first extracts all available Github URLs from the F-Droid '*index.xml*'.

```
1  // Extract List of all applications
2  XmlNodeList allApps =  doc.SelectNodes("/fdroid/application");
3
4  // Parse each application node
5  foreach (XmlNode node in allApps)
6  {
7    // Make new Dataset and add ID + Source Url.
8    AppData data = new AppData();
9    data.ID = node["id"].InnerText;
10   data.Source = node["source"].InnerText;
11
12   // Add to List.
13   appDataList.Add(data);
14 }
15
16 // Extract all usernames and repo names
17 foreach(AppData data in appDataList)
18 {
19   if (data.Source.Contains("github")) // Has Github Source
20   {
21     data.Source = data.Source.Replace("https://github.com/", "");
22     string[] urlComponents = data.Source.Split('/');
23     data.Owner = urlComponents[0];
24     data.RepoName = urlComponents.Length > 1 ? urlComponents[1] : "N/A";
25   }
26   else // NOT GITHUB REPO.
27   {
28     data.Owner = "N/A";
29     data.RepoName = "N/A";
30   }
31 }
```

Listing 1: The tool first parses the XML index file for source URLs containing "github". It then splits them and extracts username and repo name.

Not all projects have a github source, and some contain malformed URLs. In case there is no such URL, the tool will not attempt to extract anything, as the metadata analysis is restricted to Github. The splitting of the URL from lines 21-24 is done because Octokit does not use URLs but username and repo-name.

[5] https://github.com/ytrehorn/FDroidParser

The tool then iterates over all pairs and asynchronously calls a function that handles the necessary Octokit calls.

```
// Parse each individual App
foreach(AppData data in appDataList)
{
  getContributorsAsync(data).Wait();
  Console.WriteLine("Now processing App Nr. " + finalList.Count);
  File.WriteAllLines(rootpath + "/output.txt", dataAsStringList);
}

...

static async Task getContributorsAsync(AppData data)
{
  if(data.Owner == "N/A" || data.RepoName == "N/A") // Something went wrong or not Github Repo.
  {
    Console.WriteLine("Not a Github repo.");
    data.Authors = "N/A";
    finalList.Add(data);
    return;
  }

  // This starts a GitHubClient.
  var client = new Octokit.GitHubClient(new ProductHeaderValue("ytrehorn"));

  // OAuth Token authentification. Necessary to have more than 60 requests/h. With token 5000 per hour
      .
  var tokenAuth = new Octokit.Credentials(<Token String>);
  client.Credentials = tokenAuth; // Save token into current client session.

  // Read Only List of all the contributors.
  IReadOnlyList<RepositoryContributor> contributorList;

  try // Tries to read Contributors. If repo doesn't exist or no contributors available, will throw an
      exception.
  {
    contributorList = await client.Repository.GetAllContributors(data.Owner, data.RepoName);
  }
  catch(Exception e) // Catch exception and log it in data.Authors. This will then show in the
      output.
  {
    data.Authors = "Error: " + e.Message + ".";
    finalList.Add(data);
    dataAsStringList.Add(data.ID + "; " + data.Owner + "; " + data.RepoName + "; " + data.Authors);
    return;
  }

  ...
```

Listing 2: After getting all necessary user/repo-name pairs, the app then iterates over all of them and asynchronously requests the necessary information from Github using the Octokit API.

The Octokit client authenticates using an OAuth token, which allows the user to do 5000 requests per hour. All the contributors of the project are then loaded into `contributorList` and this list is then iterated over. The data extracted is in the end saved into a list of `AppData`, which is a custom class containing the app ID, repo owner, repo name and a list of all contributors.
Due to malformed URLs or wrong repo names, there had to be a few try/catch blocks. Should an extraction fail, the reason for it is logged in the output file.

In the end, all project data that was possible to extract is saved in an Excel file.

Figure 3.3 shows the relationship between the number of contributors assigned to a project and its security smells. For example, all the apps maintained by 42 contributors that suffer on average from two security smells would appear in the second last bar with the line on two of the secondary vertical axis. We see that most apps are maintained by two contributors, followed by projects developed by individuals. There are fewer projects with many contributors than projects with only a few contributors. According to the plot, small teams tend to be more capable of building projects resistant to most security code smells. RQ_1 can therefore be answered with the clear trend line shown in Figure 3.3. Large team sizes have a tendency to produce more smells on average than smaller ones, albeit the difference is rather small. It could be discussed whether the data point for projects with more than 60 contributors is statistically relevant or not, as the sample size is small.

To further examine the effect that contributors have on projects, it was important to see if a developer that contributed to multiple projects made the same mistakes in more than one project. To do this, we cross referenced the data gathered by the C# tool and a dataset from an earlier work using a relational database and Microsoft Excel[6]. From this section, the data listed each individual contributor per project. The other dataset contained a database of the number of smells per category per project. Due to time constraints it was not possible to assign responsibility for a smell to a developer. Therefore we assumed that all contributors to a project were equally responsible for a smell. The result was a list of contributors and how many smells were found in each smell category in their projects. All contributors that only worked on one project were then excluded. Similarly only smells that were present in the developer's projects were counted, zero values were ignored.

[6]This work could also have been done using *CommunityExplorer*[15], to show the relation between contributors and their projects with regard to smells.

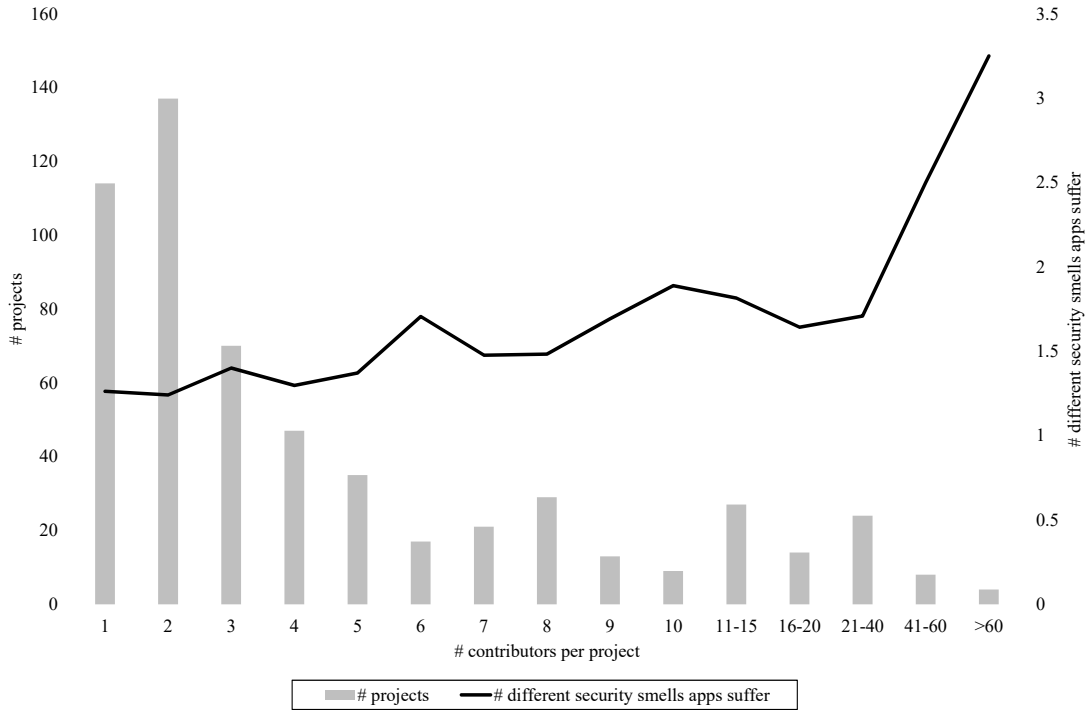


Figure 3.3: Partitioning contributors by number of different security smells

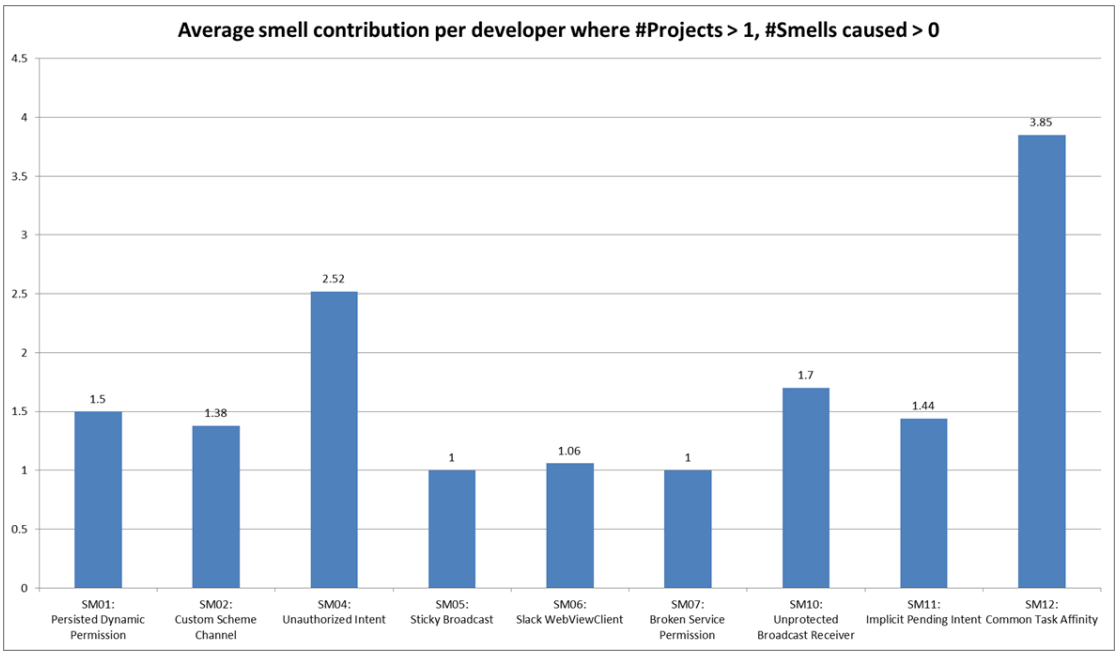


Figure 3.4: This figure shows for each smell category the number of average projects infected with a smell, if a developer participated in more than one project and caused that specific smell.

Two developers were out of the norm and had contributed to 120 and 125 projects respectively. One of them was an official F-Droid contributor. The other one is an avid open-source developer that fixed small mistakes here and there and provided a lot of translations for apps. As their values strongly deviated from the norm, they were not considered to be relevant to the statistics.

The resulting data can be seen in Figure 3.4. It was apparent that particularly SM04 and SM12 were common mistakes that developers made. Most developers don't seem to know the importance of not using task affinity where it isn't needed or that implicit intents should be avoided. The smells not shown on the graph did not appear in any projects in this dataset (1210 developers with affiliations to more than 1 project).

3.2.2 App Updates

To look into RQ_2, we investigated the smell occurrences in subsequent app releases. We inspected apps in the dataset with several releases manually. Using the linting tool, we noted the number of smells detected, and we noted if the number increased or decreased with respect to previous versions. Of the 732 projects, 33 (4%) of them released updates that resulted in a change of the total amount of smells. Many of the updates targeted new functionality, *e.g.,* addition of new implicit intents to share data with other apps, implementation of new notification mechanisms for receiving events from other apps using implicit pending intents, or registration of new custom schemes to provide further integration of app related web content into the Android system. We believe this to be due to developers focusing on new features instead of security.

For the majority of the app updates that introduced new security smells, we found the dominant cause for decreased security to be the implementation of new ICC functionality, *i.e.,* social interactions or data sharing. Hence, developers should be particularly cautious when integrating new functionality into an app.

3.2.3 Influence of Project Age and Activity

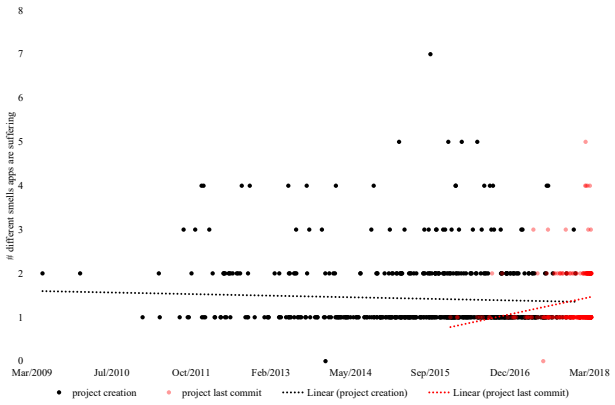
Besides the influence of updates themselves, another aspect to be considered was the frequency at which the project received new commits. To see the effects of this, we evaluated the type of ICC and two categories from the lint report (namely "security" and "correctness") based on the time since the last commit. A related question arises from the age of a project, *i.e.,* are mature projects more secure than recent ones? We investigated these two questions based on available GitHub metadata, extracted as described in subsection 3.2.1, and then related the commit dates to the issues found for the project. We extended the C# tool to extract all commits for each project including the following data:

- Timestamp

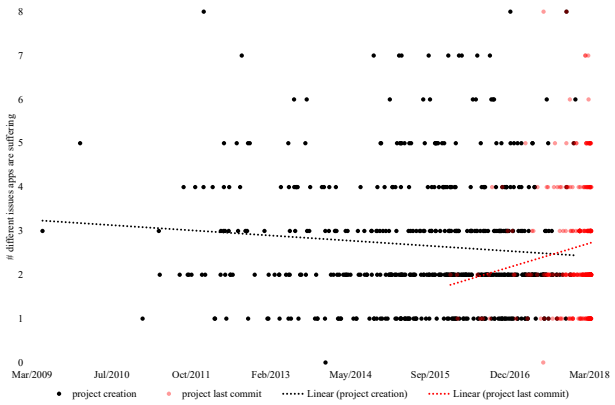
- Author
- Commit message

This data was then saved in a MongoDB database.

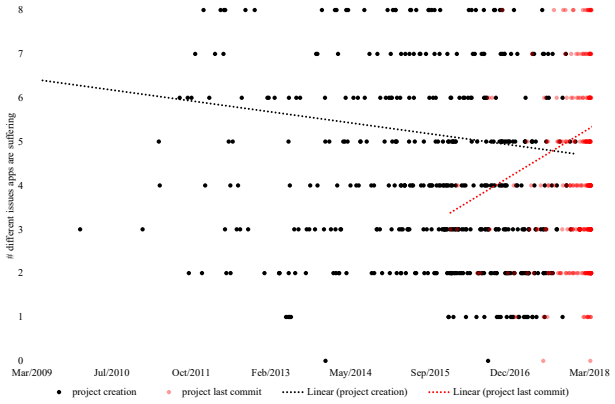
In Figure 3.5 we can clearly see a correlation between both the creation date, and the date of the last commit to the overall issue count in every plot. Especially the "correctness" category shows strong evidence that mature projects have more security issues than recent ones. We assume that this is caused by the less comprehensive checks that older IDEs performed on the source code. Similarly, apps that frequently introduce changes, *i.e.,* receive updates, are prone to have more issues. The linear trends (dotted lines) for ICC security smells and Lint security are very close in terms of elevation, and are a further indicator for the correlation of the original Android lint reports and the ones generated by the tool used in this thesis.



(a) Relation of dates to our ICC security smells



(b) Relation of dates to Lint security



(c) Relation of dates to Lint correctness

Figure 3.5: GitHub project creation and last commit date in relation to each project's issues count

3.3 Manual Analysis

We manually analyzed a portion of apps to assess how reliable the linting tool is in detecting security vulnerabilities.

We selected the top 100 apps with most smells in accordance with the ICC security smell list, and recorded the observations in a spreadsheet. We evaluated the smells reported by the tool against a vulnerability benchmark.

3.3.1 Tool Performance

To evaluate the performance of the tool, we selected the tool's proposal of smells and the proposals from two study participants. The author was one of the participants, the other was a more experienced developer.

Both participants evaluated all smells detected by the tool. The ground truth is considered to be the union of the evaluation results of participants A and B. Quite high rates of agreement were obtained between the two participants and the tool, especially for SM02, SM03, and SM05, as shown in Figure 3.6. The figure does not show all smells, since not all of them were present in great enough numbers in the 100 apps to show up in the plot. Some of the smells were not present at all. The participants tended to interpret diversely the threat caused by the *Unauthorized Intent* smell. We assume this to be caused by the very complex and flexible implementation that has been provided by Android. As expected, we found false positives. Despite the Lint failures, false positives were frequently caused by the lack of context, *e.g.,* unawareness of data sensitivity, or custom logic that mitigates the smell. For example, the tool was unable to verify custom web page white-listing implementations for `WebView` browser components, which would actually improve security.

3.3.2 Common Security Smells

This section gives a brief overview over the most common security smells and how they are usually manifested. We found, for example, that some apps were using `shouldOverrideUrlLoading` without URL white-listing to send implicit intents to open the device's default browser, rather than using their own web view for white-listed pages, thus fostering the risk of data leaks. Another discovery was the use of regular broadcasts for intra-app communication. For these scenarios, developers should solely rely on the `LocalBroadcastManager` to prevent accidental data leaks. The same applies for intents that are explicitly used for communication within the app, but do not include an explicit target, which would similarly mitigate the risk of data leaks. Moreover, unused code represents a severe threat. Several apps requested specific permissions without using them, increasing the impact of potential privilege escalation attacks.

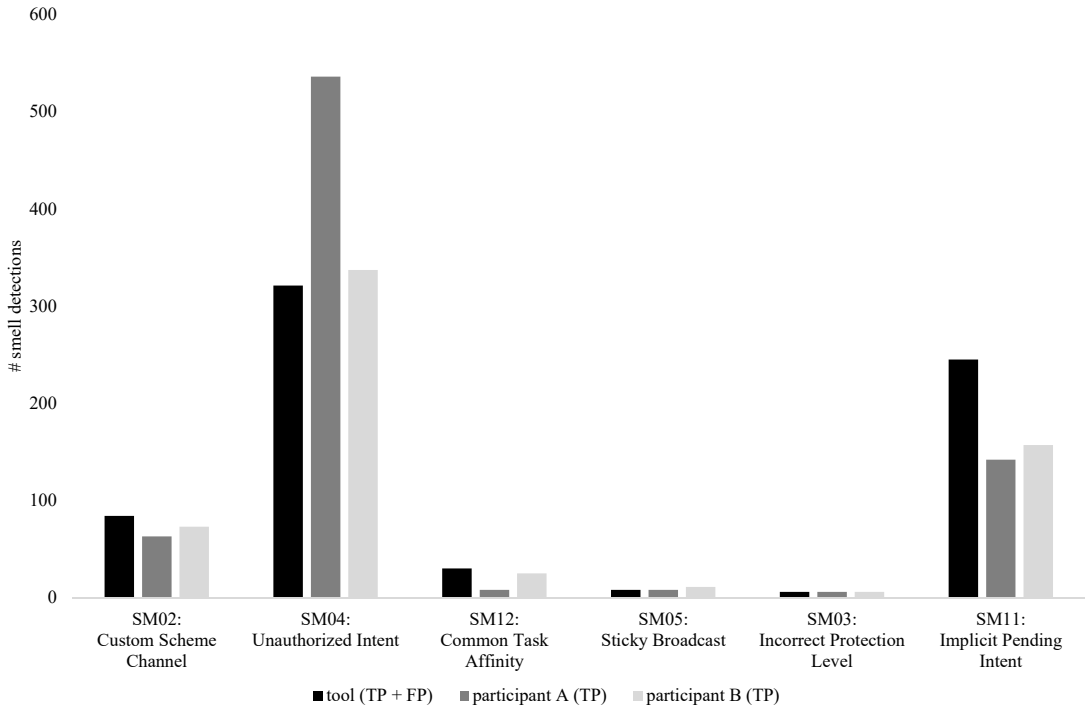


Figure 3.6: The tool's proposal of number of smells along with the proposals of the two participants for 100 apps.

In conclusion to the remaining reports of the two reviewers, the tool was able to correctly detect the security risk in 48% of cases, which is mainly due to the fact that discerning data sensitivity is non-trivial.

3.4 Unreported smells

The linting tool analyzes a project's source code and presents security smells in a report. The smells not present in the report of a project are *unreported smells* in this app. If a smell is not reported by the tool, it has until now been assumed that this smell is not present. To ensure that no large numbers of smells that were actually present went undetected, thereby making all other results unsure, we did a study on the smells not reported by the tool. We analyzed 99 apps manually in order to answer RQ_5. There are three possibilities when a smell is absent in an app: the API is not used, the API is used securely or the tool failed to identify the smell. If the API is not used, it means the prerequisites for the related smell are not present. If it is used, the smell could have been present but the tool has failed to detect it. Otherwise the API has been used securely.

3.4.1 Procedure

We used regular expressions in order to analyze the absence or presence of related APIs within these apps, after the initial evaluation of whether each smell in the app is reported or not were done by the tool. In this analysis we do not consider false positives from the tool, but focus on the true or false negatives.

For the manual analysis the following terminal commands with regular expressions were used to check if the API was not used

- **SM01 - Persisted Dynamic Permission**
  *awk 'NR==FNR{a[$0]++;next} $0 in a{delete a[$0]}END{for(x in a)print x}' <(grep -rl \.grantUriPermission *) <(grep -rl \.revokeUriPermission *)*
  This small shell script prints any occurences where the granting amount of grantUriPermission is not matched by the amount of revokeUriPermission. This however also shows an empty result if there are no pairs at all. Additionally it requires a manual check to see if single cases of the granting are present and if unmatching is done in the same file.

- **SM02 Custom Scheme Channel**
  *grep -irn -E 'android:scheme|IntentFilter\.addDataScheme' **
  We checked android:scheme in the manifest file manually to see if it was an allowed scheme or not. The addDataScheme call was checked similarly.

- **SM03 Incorrect Protection Level**
  *grep -irn \<permission **
  This regular expression checks to see if permissions are present, but we still needed to check if the permission had the correct protection level. We found the Android manifest files to have a large variation of formatting and arrangement. Thus no general pattern was found that could match protectionLevel to permissions reliably.

- **SM04 Unauthorized Intent**
  *grep –include *.java -E -rn 'startActivity|(...)|sendOrderedBroadcastAsUser' **
  Differentiation of implicit and explicit intents proved to be quite complex. Some developers used helper functions to fill the constructor arguments, some declared the intents in a line above the function call and others broke the arguments up into multiple lines, making the regular expression fail due to a undefined amount of whitespace characters being present. Evaluating whether the smell was mitigated or present proved to be complex due to these reasons.

- **SM05 Sticky Broadcast**
  *grep -rn -E 'sendStickyBroadcast|(..)|removeStickyBroadcastAsUser|BROADCAST-_STICKY' **
  The API was never used, which was also considered to be the mitigation.

- **SM06 Slack WebViewClient**
  *grep –include .java -rn ṡetWebViewClient **
  Few apps include this smell. Automation is made difficult due to the mitigation being complex to detect. Data gathered from regex (line number and filename) is not sufficient to make an informed decision about this smell.

- **SM07 Broken Service Permission**
  *grep –include \AndroidManifest.xml -E -rn 'android:exported|<service' **
  The detection if the API is used or not depended on whether the service is exposed. This was done by a check of <service> in the manifest and looking for the *exported* attribute. As formatting in the manifest file varied a lot (sometimes these two tags would be separated by a lot of lines containing either comments or other related tags) it was sometimes unclear if an *exported* tag belonged to a service or another android component. Similarly, checking for mitigation had to be done manually, as the caller and callee were often in different files and were connected in non-trivial ways.

- **SM08 Insecure Path Permission**
  *grep –include *.java -rn UriMatcher\.match **
  This smell was never present.

- **SM09 Broken Path Permission Precedence**
  *grep -rn pathpermission **
  This smell was never present.

- **SM10 Unprotected Broadcast Receiver**
  *grep –include .java -rn ṙegisterReceiver **
  The regular expression checks for presence of relevant API method. Checking for arguments of permission proved to be difficult to catch all occurrences, as developers tended to format their code in different ways both syntactically and semantically. For example one developer masked all permission arguments in a helper method that returned the type of permission, prompting a manual examination of the code. Others added comments after arguments that were followed by line breaks, making it more complex to differentiate what was part of the method call and what wasn't.

- **SM11 Implicit Pending Intent**
  *grep –include *.java -E -rn 'IntentġetActivity(|Intent.getBroadcast(|(...)|Intent.getForegroundService(' **
  Similar to SM04, differentiating between explicit and implicit intents proved to be difficult, as developers tended to have radically different ways of calling the function and declaring the contained intents.

- **SM12 Common Task Affinity**
  This smell was always present, therefore not of interest.

3.4.2 Results

As we can see from figure 3.7, most of the smells are not found because the API is not used. The figure can be a bit misleading, since not all the apps were lacking all the smells. SM12 is not represented in the graphic because this smell was always reported. Table 3.1 shows the numbers of apps out of the 99 where the smell in question is unreported, and it becomes clear that the reason SM04 seems the odd one out in the graphic is that it is unreported in only two of the apps.

The results found can be summarized for each smell:

- SM01: This smell was unreported in most of the apps, and in all cases, we used a regular expression to find the relevant API and it was not found. None of these apps are then in danger of this smell, and it is safe to say that the lack of reporting is warranted.

- SM02: Here the recommended mitigation of the smell is to not use the API, so these two categories coincide to some extent. Whether the developers are avoiding

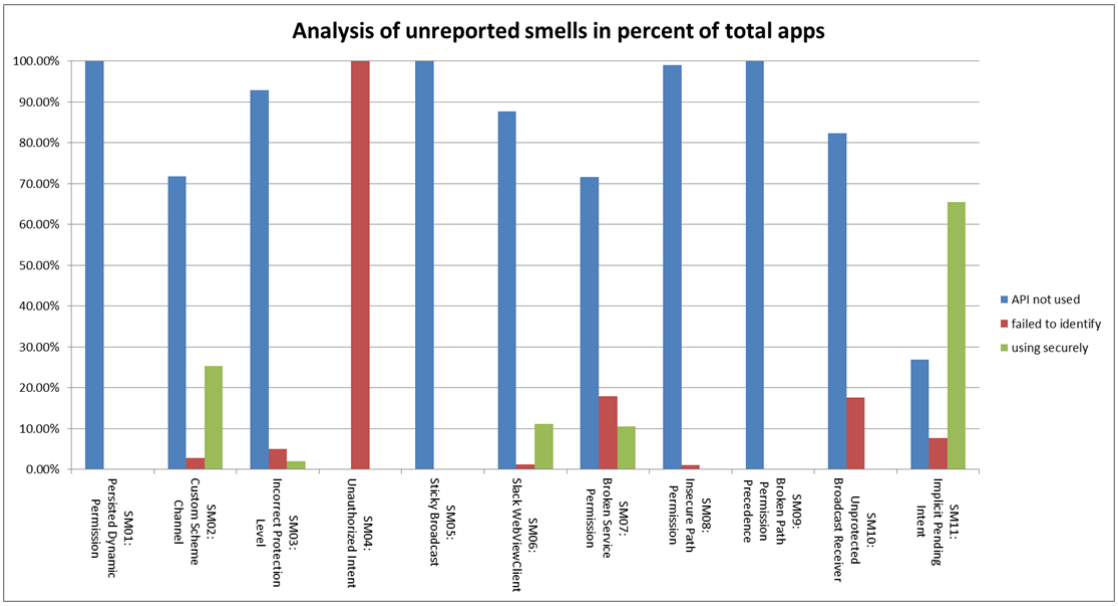


Figure 3.7: Reasons for unreported smells in 99 apps

Smell ID	Total unreported smells
SM01	96
SM02	71
SM03	99
SM04	2
SM05	88
SM06	81
SM07	95
SM08	99
SM09	99
SM10	17
SM11	52

Table 3.1: Smells unreported (both unused and undetected) in number of apps out of 99

the smell on purpose or not, there is a very low rate of failure to identify the smell, although not perfect.

- SM03: This smell was always unreported in the app selection used for this manual analysis. The linting tool failed to detect the smell in five cases. In one case the app used custom permissions without declaring any protection levels. In four other cases, the apps only used protection level "normal" which is deemed to not be enough. Two apps mitigated the issue by using correct protection levels.
- SM04: As this smell was prevalent throughout the automatic analysis, the two unreported SM04 being present despite the failure to detect them is not surprising. The two apps in which they weren't detected was due to the relevant source code being edge cases. The formatting was non-standard which is probably the reason they were passed over.
- SM05: As this smell was not prevalent in the automated analysis, chances of it appearing in the 99 manually analyzed apps was low, and "API not used" for all of the apps was the most likely outcome. Here the recommended mitigation of the smell is to not use the API, so these two categories coincide.
- SM06: Most of the apps did not use WebViewClient at all. In only one case was it not correctly identified. This was due to the syntax being convoluted and possibly being an edge-case. Nine apps used the affected API elements securely by applying access restrictions and whitelisting websites.
- SM07: The linting tool failed to identify 17 occurrences of SM07. From our understanding, detecting the smell automatically is rather difficult, as it usually spans over multiple class/source files. Therefore the manual analysis yielded more detections, as it is more difficult for a program to interpret the semantics of source code.
- SM08: This smell was always unreported in the app selection used for this manual analysis. Here the recommended mitigation of the smell is to not use the API, so these two categories coincide. One app did use `UriMatcher.match()` which was not detected by the lint tool. Otherwise all apps did not use the API / mitigated the smell.
- SM09: This smell was also always unreported in the app selection used for this manual analysis. The recommended mitigation of the smell is to not use the API, so these two categories coincide. None of the apps used this specific component of the API.
- SM10: Three apps remained undetected for this smell. It is unclear why this was the case, as the syntax was quite clear with no discernible argument for permission.

As all of the apps used linebreaks and tabs to separate the arguments, it is possible that those interfered with the detection.

- SM11: Surprisingly, a lot of the apps with no reports of SM11 used the pending intents securely. Out of 52 apps, 34 mitigated the smell by using explicit intents. Four apps failed to be identified, this was mostly due to helper methods written by the developers obscuring the smells.
- SM12: This smell was always present in the app selection used for this manual analysis, hence it is not unreported and could not be further investigated here.

In summary, we can say that the linting tool does a sufficient job detecting most smells, as the API is not used in most cases where the smell is not detected.

3.5 Placement of smells

To answer RQ_6, we further expanded the linting tool developed by the SCG to include the package name, class name and enclosing method. The enclosing method of a smell would be the name of the method in which the smell is found, the class would be the name of the class in which this method is located, and the package name is the Java package in which this class is contained. We then linted around 1350 apps again using this tool, and we wrote a script[7] to extract the information from XML files and collect it in a csv file for analysis. The script first opens the directory containing all lint reports and loads all sub-directories. These are then iterated over to extract the app ID by removing unnecessary tags such as '_src.zip' or '_src.tar.gz'. If an XML file containing the lint report is found, it is opened afterwards (not shown in code snippet).

```
1  // Iterate over folder
2  foreach (var dir in directories)
3  {
4    Environment.CurrentDirectory = dir;
5
6    var res = Directory.GetFiles(".\\", "lint−result.xml", System.IO.SearchOption.AllDirectories);
7
8    if (res.Length == 0)
9      continue;
10
11   string[] appSplit = new string[] { "\\" };
12   string[] splitPath = res[0].Split(appSplit, System.StringSplitOptions.RemoveEmptyEntries);
13
14   string appName;
15   if (splitPath.Length > 2)
```

[7] https://github.com/ytrehorn/LintXMLParser

```
16      appName = dir.Split(appSplit, System.StringSplitOptions.RemoveEmptyEntries)[2].Replace("_src",
          "");
17    else
18      appName = dir.Split(appSplit, System.StringSplitOptions.RemoveEmptyEntries)[2].Replace("_src",
          "");
19
20    if(appName.Contains("tar.gz"))
21       appName = appName.Replace("_src.tar.gz", "");
22
23    if(appName.Contains("-master"))
24      appName = appName.Replace("-master", "");
```

Listing 3: The C# script finds all sub-folders in the main directory of the lint reports, iterates over each one and extracts the app name. After this code section it will then extract the necessary metadata from the lint report.

The XML file is then parsed for the relevant categories (*security*) and from the summary (where the additional information was placed), the information is extracted. This is done by using the XML nodes *summary* and *location*.

```
1  // Extract List of all issues
2  XmlNodeList allIssues = doc.SelectNodes("/issues/issue");
3
4  // List of all Issue Data
5  List<AppData> appDataList = new List<AppData>();
6
7  // Parse each issue node
8  foreach (XmlNode node in allIssues)
9  {
10   if (node.Attributes["category"].InnerText != "Security")
11     continue;
12
13   if (!node.Attributes["summary"].InnerText.Contains("SM"))
14     continue;
15
16   AppData data = new AppData();
17   data.AppID = id;
18   data.AppName = appName;
19   string smellID = node.Attributes["summary"].InnerText;
20   string[] smellAray = smellID.Split(':');
21   data.SmellID = smellAray[0];
22
23
24   ...
25
26
27   data.Line = node["location"].Attributes["line"].InnerText;
28   data.Package = components[1].Split(':')[1].Trim();
29   data.Method = components[2].Split(':')[1].Trim();
```

```
30    data.AffectedClass = components[3].Split(':')[1].Trim();
31
32
33    ...
34
35  }
```

Listing 4: The additional metadata is contained in the *summary* tag in the xml file.

As Smells 02 and 12 are exclusively found in the Android manifest, they were excluded in this step as the information gained from them could not be related to class, package and surrounding method.

3.5.1 In relation to surrounding methods

Figure 3.8 shows the distribution of average number of distinct surrounding methods per smell and the maximum of distinct methods for all apps.

SM01 was only found in a single method on average for all occurrences. This might be due to the relevant API having a very focused application. SM03 was also only found in a single method on average, due to low occurrence of this smell. If this smell was present, it was only once in an app. This is also reflected in the following sections regarding classes and packages. Due to low occurrence, SM01 and SM03 are only found in one class and package.

SM04 is much more widely distributed than any other smells, with the average being 2.88 distinct methods per smell and the maximum 22 different methods in which the smell appears. This is most likely due to intents being widely used in android apps. As ICC can happen in many components of applications, intents have a widespread use. Despite this, it is still surprising to see the maximum number of distinct methods being 22. The responsible app only contains 19 class files, which makes this high number of methods even more surprising.

SM05 has a rather low average due to the fact that the smell is not very widespread and is only present in 18 apps. Developers do seem to use it more than once on average, as the maximum is only two but the average still being 1.61.

SM06 has a relatively low average, despite having been detected in 72 apps in the extended analysis. Most apps have this smell in one method, with three being out of the norm. This is most likely due to the fact that web browsing within apps is only used for specific cases, whereas most outgoing web requests are usually redirected to the phones browser. A lot of apps use this in either authentication windows, Web GUIs or ReCaptcha checks.

SM07 also has a relatively low average of methods that contain the smell. This is due to most apps not using more than one service that is exposed. Do note that the sample size for this evaluation was nine apps, but only one had three services, whereas the rest had one.

SM08 and SM09 were not present in the analysis.

SM10 had a slightly higher average than all the other smells except SM04. Broadcast Receivers are used whenever an app would like to register for application intents. Due to smells pertaining to intents being more widespread, it makes sense for the average number of surrounding methods to be higher. There is a tendency that apps receive fewer intents than they send.

SM11 has an average of 1.43 distinct surrounding methods per app. It is not as high a number as SM04 dealing with direct intents, because developers seem to prefer working with immediate intents and not delayed ones. This is also reflected on the maximum number of methods being five and not as high as 22 for SM04.

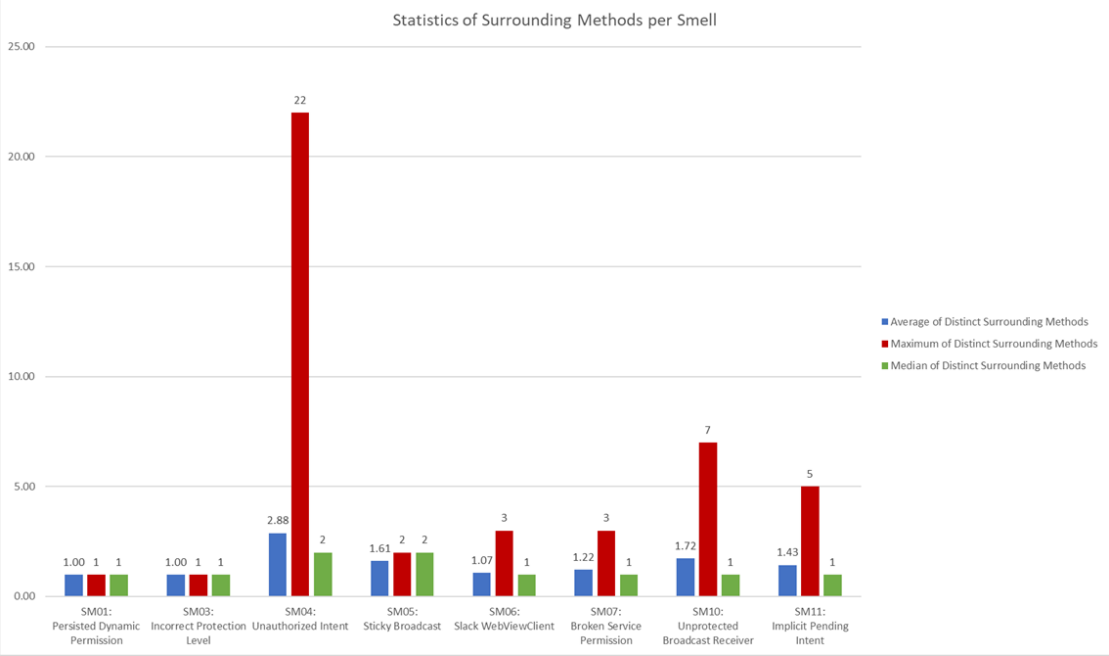


Figure 3.8: This graph shows the average number of distinct surrounding methods per smell for all apps in blue, the maximum number in red and the median in green.

3.5.2 In relation to classes

The classes containing the smells are distributed in a similar fashion as in subsection 3.5.1. Again SM01 and SM03 have low numbers due to not being present very often and in those cases only once.

SM04 shows high numbers as expected, since almost all apps use intents and most of them use them incorrectly. The average is still quite high, but the maximum is lower. SM04 in classes is less spread out than in methods, which makes sense as a class can contain a lot of methods.

SM05 has the same distribution as in distinct methods. The smell was only found in 18 apps, each method it was found in must have been in a separate class for the distribution to be identical.

SM06 has an average number of classes of 1.28, which is higher than surrounding methods. This probably points to methods being called the same across classes. A quick check shows that functions are usually called something like `run` or `onCreate`. Therefore function names are more closely related while the classes are usually not the same. This might point to either copy-pasting of code due to code working in a similar fashion, or the function of methods being the same so that it makes sense to call them the same. Either way, it is interesting to see classes having a higher average than methods.

SM07 has a lower average for classes than surrounding methods. This is a bit unexpected since the callee in this smell usually only uses the relevant functions once. Having only 11 smells to work with might make the statistics a bit uncertain. One app has this smell twice in same class but different method names. This might explain the difference. Otherwise the classes are quite similar to surrounding methods. The same app causes the maximum number for both graphs.

SM10 has the average amount of classes higher than the average amount of methods. This is due to methods having generic names such as `onCreate`, `onResume` or `registerXReceiver`. The maximum is quite a bit higher than for methods, which also reflects the above mentioned relationship in the "largest" app.

SM11's average is slightly higher than the one for methods. This is again due to the fact that these methods have generic names used in different classes. This can also be seen in the maximum.

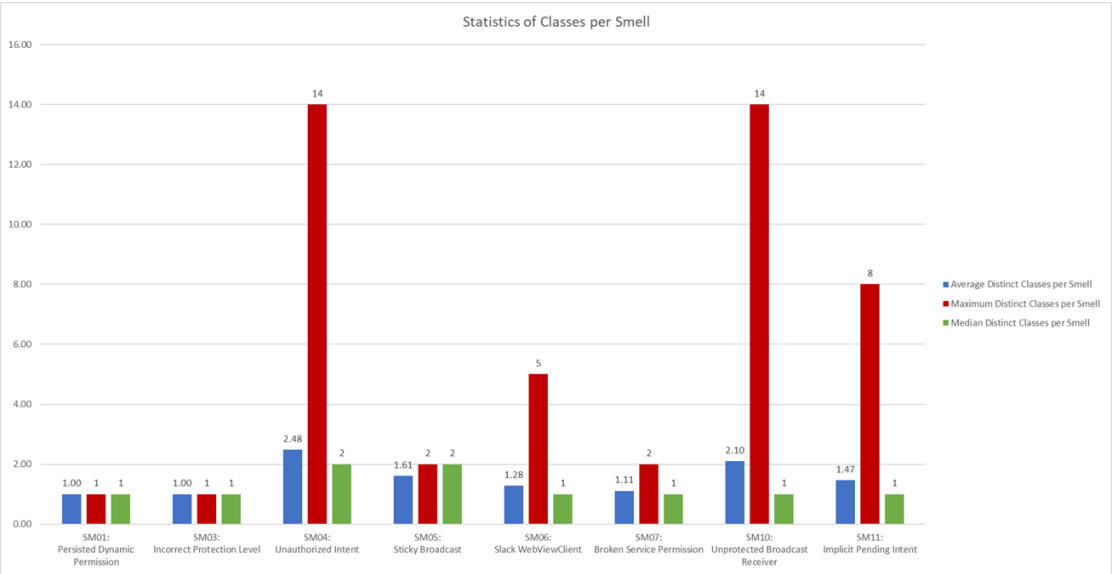


Figure 3.9: This graph shows the average number of distinct classes per smell for all apps in blue, the maximum number in red and the median in green.

3.5.3 In relation to packages

Again SM01 and SM03 are as before.

SM04 has high numbers as usual. Average packages is lower than classes which is to be expected, as packages contain multiple classes which contain multiple methods, so there should be less packages than classes. One app, however, manages to have a lot of small packages. Also, this is not the same app as in the previous two plots, which only has one package.

SM05 has the same distribution as in distinct methods. This is a bit surprising and it would seem the methods are each found in separate packages.

SM06 has an average that is in between the one for methods and the one for classes. It is good that the average is smaller than for classes, as it is excessive to have too many packages. Methods is still lowest due to generic naming. A lot of apps tend to have one package name per Android component. This means that this lower average points to the smells being mostly in one package per app. The maximum app has three packages, but 12 smells detected. Some developers have one package for the entire app, also keeping it low.

SM07 shows the same numbers as in classes, lower than surrounding methods. This is due to one app having the smell twice in one class. Other than that the smell appears usually once per app and then classes, package and method coincide.

SM10 has a very high maximum, which is surprising. This is due to one app having very small packages. Average amount of distinct packages per smell is lower than methods or classes. Despite the generic method names for this smell, this points to the smell usually being concentrated in 1 or 2 packages. The specific component for this smell is component receivers, developers seem to concentrate these in few packages.

SM11 has the average number of distinct packages lower than methods or class. This points to the smell being concentrated in (usually) one package.

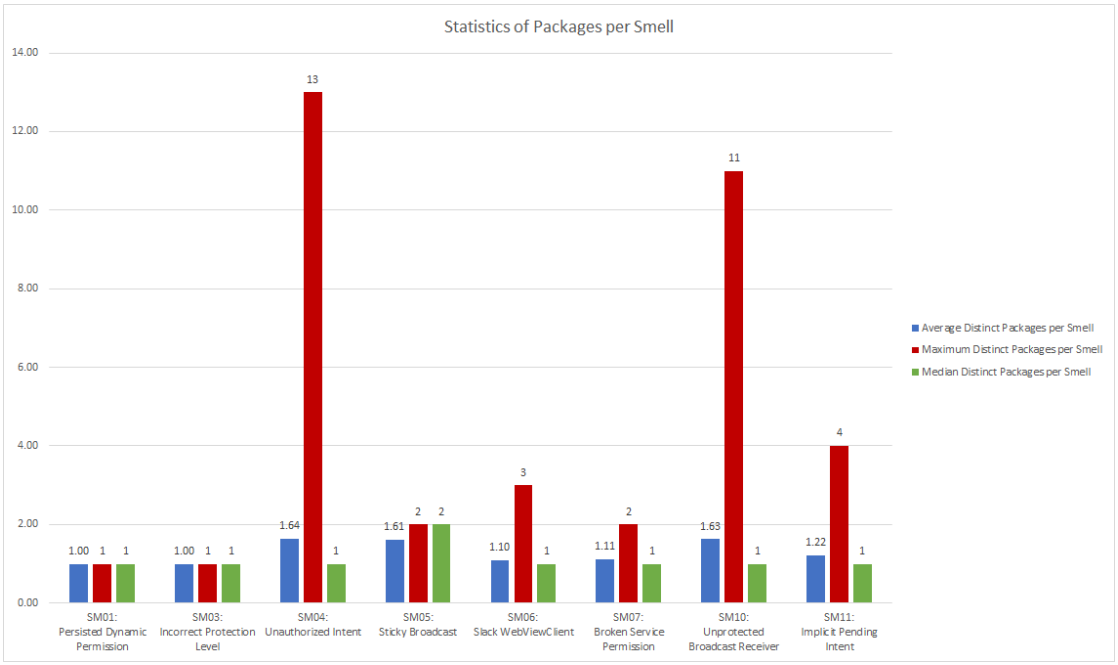


Figure 3.10: This graph shows the average number of distinct packages per smell for all apps in blue, the maximum number in red and the median in green.

3.5.4 Comparison of all three factors

As can be seen from Figure 3.11, the average of distinct packages seems to be lowest for (almost) all smells, which makes sense as packages encompass a lot of source code.

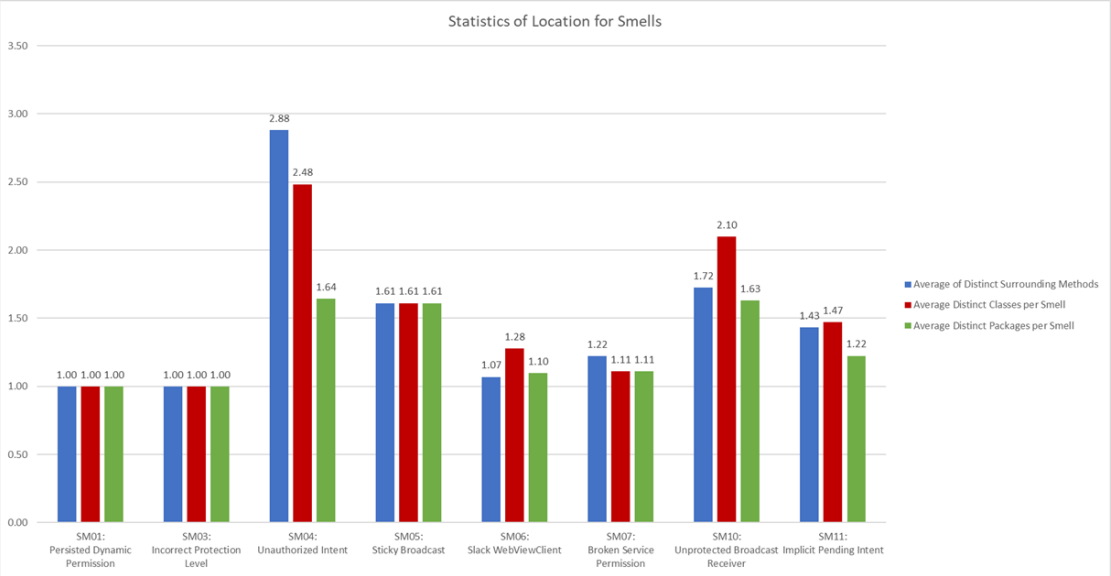


Figure 3.11: Comparing the three average distinct values.

4
Conclusions

This thesis reviewed the previously developed linting tool and added more information to the findings from the large dataset in the automated analysis. The analysis was applied to a corpus of more than 700 open-source apps. We found that larger development teams lead to more security smells on average. Small teams usually build software resistant to more security code smells than bigger teams, and fewer than 10% of apps suffer from more than two ICC security smells. This is most likely due to the fact that it becomes harder to get an overview of the whole source code, as authors usually do not work on the entire codebase. Furthermore, app updates rarely added new security issues and if they did, it was updates that add new ICC functionalities. Thus developers have to be very careful about integration of new functionality into their apps.

We found that more mature projects tend to have a higher amount of security issues than more recent ones, except for apps that are updated frequently, for which that effect is reversed. It is advisable for developers of long-lived projects to continuously update their IDEs, as old IDEs have only limited support for security issue reports, and therefore countless security issues could be missed.

A manual investigation of 100 apps shows that the linting tool successfully finds many different ICC security code smells, and about 48% of them in fact represent vulnerabilities, thus it constitutes a reasonable measure to improve the overall development efficiency and software quality.

Security aspects such as secure default values and permission systems are recommended to be considered in the initial design of a new API, since this would effectively mitigate many issues like the very prevalent Common Task Affinity smell.

We analyzed the existence of ICC security smells in apps, and did a study of their

absence, finding that the absence is mostly due to the prerequisites for the smell not being present.

Finally a study of the placement of smells in apps found that some apps have very strongly distributed smells, but these also tend to have a lot of smells. Normally one can expect the smells to be distributed in 1-2 methods, classes and packages. The hierarchy of "size" is normally Method > Class > Package, but for generic method names one sometimes gets Class > Method.

5
Threats to validity

One important threat to validity is the completeness of the study leading to the development of the linting tool, *i.e.,* whether all related papers in the literature could be identified and studied.

Only benign apps were interesting for the research, as in malicious ones it is unlikely that developers will spend any effort to accommodate security concerns. Thus, just apps that were available on GitHub and the F-Droid repository were collected. However, the dataset may still have malicious apps that evaded the security checks of the community or the market.

The existence of security smells in the source code of an app was analyzed, whereas third-party libraries could also introduce smells.

This analysis is intra-procedural and suffers from inherent limitations of static analysis. Moreover, many security smells are in fact true smells only if they deal with sensitive data, but this analysis cannot determine such sensitivity.

6
Related work

Reaves *et al.* studied Android-specific challenges to program analysis, and assessed existing Android application analysis tools. They found that these tools mainly suffer from lack of maintenance, and are often unable to produce functional output for applications with known vulnerabilities [18]. Li *et al.* studied the state-of-the-art work that statically analyses Android apps [13]. They found that much of this work supports detection of private data leaks and vulnerabilities, a moderate amount of research is dedicated to permission checking, and only three studies deal with cryptography issues. Unfortunately, much state-of-the-art work does not publicly share the concerned artifacts. Linares-Vasquez *et al.* mine 660 Android vulnerabilities available in the official Android bulletins and their CVE details,[1] and present a taxonomy of the types of vulnerabilities [14]. They report on the presence of those vulnerabilities affecting the Android OS, and acknowledge that most of them can be avoided by relying on secure coding practices. Finally, Sadeghi *et al.* review 300 research papers related to Android security, and provide a taxonomy to classify and characterize the state-of-the-art research in this area [20]. They find that 26% of existing research is dedicated to vulnerability detection, but each study is usually concerned with specific types of security vulnerabilities.

Some research is devoted to educating developers in secure programming. Xie *et al.* interviewed 15 professional developers about their software security knowledge, and realized that many of them have reasonable knowledge but do not apply it as they believe it is not their responsibility [27]. Weir *et al.* conducted open-ended interviews with a

[1] `http://cve.mitre.org` — Common Vulnerabilities and Exposures, a public list of known cyber-security vulnerabilities.

dozen app security experts, and determined that app developers should learn analysis, communication, dialectics, feedback, and upgrading in the context of security [24]. Witschey *et al.* surveyed developers about their reasons for adopting or not adopting security tools [25]. Interestingly, they found the perceived prestige of security tool users and the frequency of interaction with security experts to be important for promoting security tool adoption. Acar *et al.* suggest a high-level research agenda to achieve usable security for developers. They propose several research questions to elicit developers' attitudes, needs and priorities in the area of security [1].

Numerous researchers have dedicated their work to detecting common ICC vulnerabilities. Despite the fact that their expression has changed over time, the vulnerability classes have remained largely the same. Chin *et al.* discuss the ICC implementation of Android and examine closely the interaction between sent and received ICC messages [5]. Despite the fact that their work is based on a small corpus containing only 20 apps, they were able to detect various denial-of-service issues in numerous application components, and conclude that the message-passing system in Android enables rich applications, and encourages component reuse, while leaving a large potential for misuse when developers do not take any precautions.

Felt *et al.* discovered that permission re-delegation, also known as confused deputy or privilege escalation attack, is a common threat, and they pose OS level mitigations conceptually similar to the same origin policy in web browsers [6]. The community aimed on the one hand for preciseness, as countless tools to detect these flaws in ICC have been released, notably Epicc [17] and IccTA [12] with a significantly improved precision. On the other hand, the app coverage began to play a major role, as in the work of Bosu *et al.* who recently discovered with their tool inadequate security measures, including privilege escalation vulnerabilities, among inter-app data-flows from 110 000 real-world apps [4].

Along with passive analysis, active countermeasures and attacks began to emerge in the scientific community. Garcia *et al.* crafted a state-of-the-art tool to automatically detect and exploit vulnerable ICC interfaces to provoke denial-of-service attacks amongst others [8]. They identified exploits for more than 21% of all apps appraised as vulnerable. Xie *et al.* presented a bytecode patching framework that incorporates additional self-contained permission checks avoiding privilege issues during runtime, generating a remarkably low computational overhead [28]. Ren *et al.* successfully investigated design glitches in the multitasking implementation of Android, uncovering task hijacking attacks that affected every OS release and were potentially duping user perception [19]. They considered in particular the taskAffinity and taskParentReparenting attributes of the manifest file that allow views to be dynamically overlaid on other apps, and provided proof-of-concept attacks. Wang *et al.* assessed the threat of data leakage on Apple iOS and Android mobile platforms and show serious attacks facilitated by the lack of origin-based protection on ICC channels [23]. Interestingly, they found effective

attacks against apps from such major publishers as Facebook and Dropbox, and more importantly, indicate the existence of cross-platform ICC threats. Researchers have found interest in reinforcing the Android ICC core framework. Khadiranaikar *et al.* propose a certificate-based intent system relying on key stores that guarantee integrity during message exchanges [11]. In addition to securing the ICC-based communication, Shekhar *et al.* proposed a separation of concerns to reduce the susceptibility for manipulation of Android apps, by explicitly restricting advertising frameworks [21]. Ahmad *et al.* elaborated on problematic ICC design decisions on Android, and found that missing consistent message types and conformance checking, unpredictable message interactions, and a lack of coherent versioning could break inter-app communication and pose a severe risk [2]. They recommend a centralized message-type repository that immediately provides feedback to developers through the IDE.

In summary, existing studies often dealt with a specific issue, whereas the research that this paper is part of covers a broader range of issues, making the results more actionable for practitioners. Moreover, previous work often overwhelms developers with many identified issues at once, whereas the developed tool provides feedback during app development where developers have the relevant context. Such feedback makes it easier to react to issues, and helps developers to learn from their mistakes [22].

7

Anleitung zu wissenschaftlichen Arbeiten

The following section describes how to securely use the ICC API. As identified in chapter 2, there are numerous security code smells that can potentially threaten Android applications. Ergo, it is important for Android developers to avoid unsafe API usage, as neglecting to do so opens the app to malicious activities. This chapter will deal with the following topics

- What the security smell is.
- Which part of the related API to avoid.
- How to mitigate the smell.

This will be shown using code snippets with correct usage when pertinent.

7.1 Identifying security smells and how to avoid them

7.1.1 SM01 - Persisted Dynamic Permission

In Android, one can access protected resources using a Uniform Resource Identifier (URI), which is granted at runtime.

Security Issue

This sort of access to protected resources is meant to be granted temporarily and then be revoked again. However, it often happens that a developer forgets to revoke the permission again, which keeps the access to the resource active. This leads to the recipient of said access to have long lasting access to potentially sensitive data.

Symptom

In the code, the developer grants access via `Context.grantUriPermission()`, but no corresponding `Context.revokeUriPermission()` is present.

Mitigation

Developers should be vigilant to ensure that access is revoked again after it has been granted. Always check if there is a corresponding revoke call to a grant. Another way of mitigating this issue is to attach the relevant data to the intent instead of using a URI.

7.1.2 SM02 - Custom Scheme Channel

Instead of standard URIs such as *http://*, Android allows the registration of custom URIs such as *myapplication://* throughout the operating system. This could for example be used for the app to register an activity to respond to the designated URI via an intent filter in the android manifest. This would look as follows

```
<activity android:name=".MyUriActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="myapp" android:host="path" />
  </intent-filter>
</activity>
```

Listing 5: Example of using custom schemes for activities.

Users can then access the associated activity by opening specified hyperlinks in a wide set of applications.

Security Issue

Any application is potentially able to register and use any custom schemes set and used by other apps. This is unwanted behaviour, e.g. a malicious app could access URIs that contain access tokens or similar credentials, without a possibility for the caller to identify such leaks.

Symptom

The attribute `android:scheme` exists in the `intent-filter` nodes in the android manifest file. Another possibility is the use of `IntentFilter.addDataScheme()` within the sourcecode.

Mitigation

It is important to never send any sensitive data such as access tokens or credentials via such URIs. Instead of custom schemes, one should use system schemes that have proper restrictions on the intended recipients.

7.1.3 SM03 - Incorrect Protection Level

Android apps must request permission to access sensitive resources. In addition to that, custom permissions can be set by developers to limit the range of access for specific features, based on the protection level that is set for other applications. This is done in the android manifest file using `android:protectionLevel`. Permissions can for example be set like this

```
<permission android:name="android.permission.SET_ACTIVITY_WATCHER"
    android:label="@string/permlab_runSetActivityWatcher"
    android:description="@string/permdesc_runSetActivityWatcher"
    android:protectionLevel="signature" />
```

Listing 6: Example of permissions in the android manifest.

The permission is set in `android:protectionLevel` and has a setting of `"signature"` in this example. There are three levels: `"normal"`, `"dangerous"` and `"signature"`, with signature being the strictest. According to the android documentation, signature is "A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.". It is important to protect sensitive data with high enough protection levels.

Security Issue

An app that declares a new permission may neglect the selection of the correct protection level, *i.e.,* a level whose protection is appropriate to the sensitivity of the resource. Any protection level that is too low might allow easier access to sensitive data.

Symptom

Any custom permission which is missing the correct `android:protectionLevel` attribute in the manifest file.

Mitigation

It is important to protect sensitive features/resources with adequate protection levels. In most cases this should at least be `"dangerous"`, ideally `"signature"`.

7.1.4 SM04 - Unauthorized Intent

Android application components can communicate with other Android applications. This connection is based on an Intent object. With Intents one can send one-way requests, such as requesting an image from a gallery, or sending text messages. Intent receivers have the ability to request custom permissions that clients need to have, before they are allowed to communicate with it. These intents and receivers are then *protected*.

Intents come in two flavors, *explicit* and *implicit.*

```
1 // Example for explicit intent
2 Intent i = new Intent(this, ActivityTwo.class);
3 i.putExtra("Value1", "This value one for ActivityTwo ");
4 i.putExtra("Value2", "This value two ActivityTwo");
5
6 // Example of implicit intent
7 Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.unibe.ch"));
8 startActivity(i);
```

Listing 7: Example of the two kinds of intents.

The main difference is that in line 2, the system is *explicitly* told which component should be called, whereas on line 7 it only *implicitly* says the action and its content that one wants to perform. The system will then search for components that are able to do so. Should multiple components be found, the OS will provide a dialogue box with a selection.

Security Issue

Any app has the ability to send unprotected intents, regardless of required permissions. Likewise, it can register itself to receive unprotected intents. This means, apps could escalate their privileges by sending unprotected intents to privileged targets. This could be apps that have access to elevated features such as camera access. Another issue is that malicious apps could receive unprotected implicit intents by registering to them and leak/manipulate their data.

Symptom

The existence of unprotected implicit intents. Intents that request a return value that do not check if the sender has appropriate permissions to initiate an intent. Several methods on the Context class for initiating unprotected implicit intents are present, such as

- `startActivity(...)`
- `sendBroadcast(...)`
- `sendOrderedBroadcast(...)`
- `sendBroadcastAsUser (...)`
- `sendBroadcastAsUser(...)`
- `sendOrderedBroadcastAsUser(...)`

Mitigation

Always use explicit intents to send sensitive data. When processing an intent, the contained data from other components should always be validated to make sure they are not spoofed. Additionally, adding one's own custom permissions to implicit intents might make the user more aware of the process, thus raising the level of protection.

7.1.5 SM05 - Sticky Broadcast

Broadcasts are used by the android system to convey information to other applications. Sticky broadcasts are a special type of broadcasts, as they persist after they are announced instead of becoming inaccessible.

```
1 // Example of a sticky broadcast.
2 Intent intent = new Intent("some.custom.action");
3 intent.putExtra("some_boolean", true);
4 sendStickyBroadcast(intent);
```

Listing 8: Example of how a sticky broadcast could appear in the sourcecode.

An example of this would be the battery gauge, any application can receive the current battery level regardless of when the last broadcast was.

Security Issue

Any application can watch broadcasts, and particularly sticky broadcasts receivers can tamper with them. Manipulating these sticky broadcasts can then convey false information to future recipients.

Symptom

Calls to sending sticky broadcasts appear in the code. These include the methods

- `sendStickyBroadcast(...)`
- `sendStickyBroadcastAsUser(...)`
- `sendStickyOrderedBroadcast(...)`
- `sendStickyOrderedBroadcastAsUser(...)`
- `removeStickyBroadcast(...)`
- `removeStickyBroadcastAsUser(...)`

used on the `Context` object.
Additionally, `android.permission.BROADCAST_STICKY` appears in the manifest.

Mitigation

Do not use sticky broadcasts. Instead, use non-sticky broadcasts. Use other mechanisms such as explicit intents for other apps to retrieve the current value.

7.1.6 SM06 - Slack WebView Client

Android uses `WebView` to permit web browsing within apps. By default, the Activity Manager is asked by the `WebView` to choose the proper handler for the URL. Should a `WebViewClient` be provided to `WebView`, the host application handles the URL.

Security Issue

The default implementation of a `WebViewClient` does not impose any restrictions on which websites to access. This means it can be used to open malicious websites that attack the user.

Symptom

The `WebView.setWebViewClient()` exists in the code but the `WebViewClient` instance does not apply any access restrictions in `WebView.shouldOverrideUrlLoading()` , i.e., it returns false or calls `WebView.loadUrl()` right away. Also, the issues appears if the implementation of `WebView.shouldInterceptRequest()` returns null.

Mitigation

The last thing the developer should do is provide basic input validation as follows

```
1 private class MyWebViewClient extends WebViewClient {
2    @Override public boolean shouldOverrideUrlLoading(WebView view, String url) {
3      if (Uri.parse(url).getHost().equals("www.unibe.ch")) {
4        return true;
5      }
6      return false;
7    }
8  }
```

Listing 9: Example of how a sticky broadcast could appear in the sourcecode.

The most secure way however, would be to use a whitelist of trusted websites for validation. For example the SafetyNet API [1] could be used to provide information about the website visited.

[1] https://developer.android.com/training/safetynet/safebrowsing.html

7.1.7 SM07 - Broken Service Permission

There are two ways to start a service. Either `onBind` or `onStartCommand`. The latter of the two allows a service to persist in the background, even after the client disconnects. Apps that use Android IPC to start a service may not possess the same permission as the service provider itself.

Security Issue

Should a service be exposed, the called service can be abused. If the callee (service provider) is in possession of the required permissions, the caller will also get access to the service. This could lead to a privilege escalation.

Symptom

The developer neglects to check if the caller has the right permissions to access the service. This issue is present if a caller uses `startService(...)` and the called service then uses one of the following methods

- checkCallingOrSelfPermission(...)
- enforceCallingOrSelfPermission(...)
- checkCallingOrSelfUriPermission(...)
- enforceCallingOrSelfUriPermission(...)

to verify the permissions of the request.
Any calls on the `Context` object for permission checks will then fail, as the system considers the callee's permissions instead of the component that called it in first place. The issue also appears if the following calls on the `Context` object are made

- `checkPermission(...)`
- `checkUriPermission(...)`
- `enforcePermission(...)`
- `enforceUriPermission(...)`

while additional calls to `getCallingPid(...)` or `getCallingUid(...)` on the `Binder` object exist.

Mitigation

Always check the callers permissions before performing privileged operations on its behalf by using `checkCallingPermission(...)` or `checkCallingUriPermission(...)` on the `Context` object. Ideally, do not use `onStartCommand` to start a service. Ensure that the necessary permissions to access a service have been set.

7.1.8 SM08 - Insecure Path Permission

Apps can share data with other apps. Aside from the regular permissions that apply to the entirety of a content provider, one can set path-specific permissions that are more granular. Path permission checks in the manifest file differentiates between paths with one slash or double slashes. If there is a mismatch of the two, only the permissions on the whole content provider is considered.

Security Issue

The `UriMatcher` provided by the Android framework, which is recommended for URI comparison in the `query` method of a content provider, considers single slash and double slash paths to be identical. It will always forward the request to the initially intended resource. This means that what are assumed to be protected resources, could be accessible to unauthorized apps.

Symptom

A `UriMatcher.math()` is used to validate URIs. Should this be in the sourcecode and a `path-permission` attribute is present in the manifest file, the security issue could occur.

Mitigation

As this bug still exists in the android framework, it is best to write one's own URI matcher. Do not use `UriMatcher`.

7.1.9 SM09 - Broken Path Permission Precedence

In Android's `ContentProvider`, permissions which are more granular should take precedence over those with a larger scope.

Security Issue

A bug identified in recent releases of Android causes `ContentProvider.enforceReadPermissionInner(...)` to ignore path permissions, so that they do not take precedence over permissions on the content provider as whole. This means that content providers could accidentally grant access to other apps.

Symptom

This issue appears when the content provider is protected by path-specific permissions. These are found in the manifest file with the keyword `path-permission`.

Mitigation

As long as the bug is present in new android releases, path permissions should not be used. Instead one should use a distinct content provider with dedicated permissions for every individual path.

7.1.10 SM10 - Unprotected Broadcast Receiver

A broadcast receiver is an Android component that allows apps to register for system or app events. All registers that subscribed to a certain event, are notified by the system when it happens. Static broadcast receivers are registered in the manifest file, and start even if an app is not currently running. Dynamic broadcast receivers are registered at run time in Android code, and execute only if the app is running.

Security Issue

Any app can register itself to receive a broadcast. If a broadcast receiver is unprotected in an app, it exposes itself to any other app able to initiate the broadcast. Hence, if the developer neglects to check for permissions for a broadcast receiver, data leaks or unintended behaviour can occur due to spoofed intents [16].

Symptom

If there is a call to `Context.registerReceiver(...)` without any arguments for permission in the source code, this is a symptom of this smell. One should watch out where the permission argument is missing or is `null`.

Mitigation

When registering a broadcast receiver, developers should ensure they have sound permissions. The following example shows how to use `Context.registerReceiver(...)` properly with a permission string. This string should be the same as defined in the manifest by a `<permission>` tag.

```
String PERMISSION_STRING_PRIVATE_BROADCASTER = "PERMISSION.NAME";
IntentFilter filter = new IntentFilter(ACTION_SAMPLE);
Context.registerReceiver(mReceiver, filter,
        PERMISSION_STRING_PRIVATE_BROADCASTER, null);
```

Listing 10: Example of how to register a receiver with proper permissions.

7.1.11 SM11 - Implicit Pending Intent

There are different kinds of Intents in Android. One of these is the `PendingIntent`. This type of intent differs from other ones in that the specified action is executed in the future. The intent is executed on behalf of the app that sends the intent, with the identity and permissions of this app. This happens regardless of whether the app is running or not.

Security Issue

Implicit pending intents can be intercepted by any app [16]. The app can then use the `send(...)` method of the pending intent to submit arbitrary intents on behalf of the initial sender. A malicious app can then manipulate the data of the intent and use the permission of the initial sender. Relaying of pending intents could be used for intent spoofing attacks.

Symptom

The security issue is present if there is an initiation of an implicit `PendingIntent` in the code. This means methods such as

- `getActivity(...)`
- `getBroadcast(...)`
- `getService(...)`
- `getForegroundService(...)`

are called on the `PendingIntent` object without specifying the target component.

Mitigation

One should use explicit pending intents, instead of implicit ones. This is recommended by the official documentation.[2]

[2] `https://developer.android.com/reference/android/app/PendingIntent.html`

7.1.12 SM12 - Common Task Affinity

Tasks are a collection of activities that a user interacts with when performing a certain job. Task affinity defines which activity a task wants to belong to. By default the activities in an app prefer to be in the same task and have the same affinity.

Security Issue

If apps have the same task affinities, their activities can overlap one another. This can for example be used to overlay a voice record button onto a phone call. However, this can be abused by malicious apps to hijack another app's activities. Using the same task affinity will allow for spoofing from malicious apps to occur.

Symptom

If the task affinity is not empty, this smell can be present. This means that other apps using the same task affinity can overlay their activities.

Mitigation

If the developer does not actively want to use task affinity to integrate other app's functionalities, it is best to leave the task affinity empty. Should the need to use task affinities arise, one should try to only set it for tasks that are safe to share with other apps.

Bibliography

[1] Yasemin Acar, Sascha Fahl, and Michelle Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *IEEE SecDev 2016*, 2016.

[2] Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. Inter-app communication in Android: Developer challenges. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 177–188. IEEE, 2016.

[3] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security Privacy*, 12(4):55–58, July 2014.

[4] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[6] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, page 88, 2011.

[7] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz. Security code smells in Android ICC, 2018.

[8] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671. ACM, 2017.

[9] M. Ghafari, P. Gadient, and O. Nierstrasz. Security smells in Android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 121–130, Sept 2017.

[10] Beth H. Jones and Amita Goyal Chin. On the efficacy of smartphone security: A critical analysis of modifications in business students' practices over time. *International Journal of Information Management*, 35(5):561 – 571, 2015.

[11] Babu Khadiranaikar, Pavol Zavarsky, and Yasir Malik. Improving Android application security for intent based attacks. In *Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2017 8th IEEE Annual*, pages 62–67. IEEE, 2017.

[12] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[13] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of Android apps: A systematic literature review. *Information and Software Technology*, 88:67 – 95, 2017.

[14] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on Android-related vulnerabilities. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 2–13, Piscataway, NJ, USA, 2017. IEEE Press.

[15] Leonel Merino, Dominik Seliner, Mohammad Ghafari, and Oscar Nierstrasz. Communityexplorer: A framework for visualizing collaboration networks. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 2:1–2:9, New York, NY, USA, 2016. ACM.

[16] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of Android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 43–52. ACM, 2017.

[17] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, Washington, D.C., 2013. USENIX.

[18] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. *droid: Assessment and evaluation of Android application analysis tools. *ACM Comput. Surv.*, 49(3):55:1–55:30, 2016.

[19] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in Android. In *USENIX Security Symposium*, pages 945–959, 2015.

[20] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[21] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, 2012.

[22] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. Jit feedback — what experienced developers like about static analysis. In *Proceedings of the 26th IEEE International Conference on Program Comprehension (ICPC'18)*, 2018.

[23] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM Conference on Computer and Communications Security*, 2013.

[24] Charles Weir, Awais Rashid, and James Noble. Reaching the masses: A new subdiscipline of app programmer education. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 936–939. ACM, 2016.

[25] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271. ACM, 2015.

[26] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on Android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[27] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, Sept 2011.

[28] Jiayun Xie, Xiao Fu, Xiaojiang Du, Bin Luo, and Mohsen Guizani. Autopatchdroid: A framework for patching inter-app vulnerabilities in Android application. In *Communications (ICC), 2017 IEEE International Conference on*, pages 1–6. IEEE, 2017.

[29] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure Android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.