Improving the Security of Android Inter-Component Communication

Adam Cozzette,* Kathryn Lingel,* Steve Matsumoto,* Oliver Ortlieb,* Jandria Alexander,†
Joseph Betser,† Luke Florer,† Geoff Kuenning,* John Nilles,† and Peter Reiher†
*Computer Science Department
Harvey Mudd College
301 Platt Blvd.
Claremont, CA 91711
†The Aerospace Corporation
El Segundo, CA

***Abstract*—In the Android operating system, each application consists of a set of *components* that communicate with each other via messages called *Intents*. The current implementation of Intent handling is such that developers can inadvertently write insecure code that allows malicious applications to intercept or inject Intents to steal sensitive information or induce undesired behavior. We prevented these exploits by modifying Android's Intent handling behavior to err on the side of safety except where the developer seems to explicitly specify otherwise. Additonally, we confirmed the pervasiveness of Intent vulnerabilities by analyzing the 497 most popular free applications in Android's official application market, and proved the effectiveness of our modifications by manually verifying that they closed a substantial number of the security holes we identified.**

I. Introduction

Applications in Google's Android operating system communicate via messages called *Intents*. These messages are passed between the core components of applications, and as such are a critical piece of the Android architecture. However, if not used carefully, Intents can open several security vulnerabilities.

For example, Intents can be *explicit* (addressed to a specific component) or *implicit* (without a named destination). Using an implicit Intent when the desired destination is known may enable other applications to intercept the messages, potentially leaking sensitive data. In addition, an application component which registers to receive implicit Intents becomes publicly visible by default, allowing any other application to send it Intents. These unexpected Intents can cause the application to crash or result in undesired behavior.

While these vulnerabilities can be avoided through careful coding by developers, such an expectation is unrealistic, given the open nature of the Android application market. Chin et al. [1] propose several changes to the Android architecture to mitigate these problems, and we implemented the changes suggested in two of these recommendations:

- *Intent resolution:* we modified Android's process of Intent resolution so that implicit Intents are directed to components within the same application whenever possible.
- *Component visibility:* we tightened the criteria for making components publicly visible, protecting applications from potentially harmful Intents unless they clearly indicate that they anticipate Intents from outside applications through certain properties (Sec. III).

The changes to the Intent resolution process maintained correct behavior in a random sample of applications we tested manually, but the changes to component visibility broke some functionality in a second random sample. In Section V, we discuss the defense's effectiveness and propose an extension to Chin's suggested changes that promotes much better backward compatibility.

II. Intent Vulnerabilities

Android uses a single type of Intent for both inter- and intra-application communication. As a result, developers can inadvertently involve outside applications when they intend to perform communication solely within the boundaries of a single application. These problems give rise to several notable security vulnerabilities:

Unauthorized Intent Receipt. While explicit Intents are designed for sending messages to known destinations, implicit Intents can achieve the same results, albeit less securely. We have found that many applications take this shortcut (Section IV). A malicious component can register itself to receive standard Android Intents and thus intercept implicit Intents used by many other applications. This allows an attacker to intercept another application's private data; the Intent can then be re-sent to its original destination so that neither the user nor the original sending component is aware of the attack. Alternatively, the malicious application could handle the Intent on its own, e.g. by starting its own Activity or Service, as part of a more sophisticated attack such as phishing [1].

Intent Spoofing. By default, any component with an Intent filter is "exported" by Android and made publicly visible to the entire phone. Often developers are unaware of this behavior and do not correctly handle the possibility of receiving an Intent from an unexpected source. A malicious component can take advantage of this to induce incorrect behavior by sending an Intent that the vulnerable component blindly accepts.

This vulnerability often arises when one component sends an implicit Intent to another in the same application. The receiving component will have an Intent filter, but might

978-3-901882-50-0

only be built to handle internally generated Intents. Malicious applications could inject information into the vulnerable app by sending it an appropriate Intent; the vulnerable app has no way of identifying where the Intent came from, so it will treat the malicious Intent as if it were a trusted one from the app's own internal component.

III. Improved Intent Handling

As we have seen, the Intents system leaves naïvely implemented Android apps open to attack by malicious apps on the same phone. While mitigating these vulnerabilities requires modifying Android to handle Intents more securely, it is important to achieve this in a backwards-compatible way to preserve the many applications on the market that have these weaknesses. While a complete overhaul of the system to separate intra- and inter-application communication would provide increased security, it would be almost impossible to build a backward-compatible system or to automatically upgrade existing apps to the new communication framework. Therefore, guided by Chin et al.'s paper [1], we have proposed several guidelines for how Intents should be handled.

In particular, outbound Intents should be sent to a component to within the sending application whenever possible, and should only use the current resolution rules if this fails. The intuition is that if a developer sends an implicit Intent that can be resolved by a component in the same application, she probably intended the communication to go to that component. This policy is not likely to break many applications, because if a component in the same application is *capable* of receiving the Intent, then delivering it is almost certainly a correct outcome.

To secure inbound Intents, components should only be exported if it meets one of the following criteria:

- The component explicitly declares itself exported,
- It registers to receive Intents that carry data, or
- It registers to receive one of the standard Android Intents.

This approach effectively inverts the current system, which is biased toward exporting any component that declares an Intent filter. The first criterion is a clear sign of the developer's desires; the other two are not a guarantee that the component should be public, but suggest that the developer probably intends to provide functionality to other applications on the phone.

IV. Analysis of Existing Android Applications

To study these Intent vulnerabilities and their prevalence in real applications, we downloaded the 497 most popular (as of March 2012) free Android applications. [1] Although we did not analyze any paid applications, we believe that our sample is a reasonable representation of common Android applications and their security flaws, especially given that many applications come in both free and paid flavors that share most of their code.

To study our sample, we decompiled each application's Java bytecode using `dex2jar` and `apktool`, and statically analyzed the resulting source. We also used `apktool` to extract the manifest file from each package. In our analysis we focused on two main aspects of existing Android applications: how often they use implicit Intents for internal communication, and how often components are automatically exported despite the developer's probable intentions.

A. Implicit Intents Used for Internal Communication

To investigate how frequently implicit Intents are used for internal communication, we analyzed both the manifest file and the decompiled Java source for each application. Using the Intent filter declarations in the manifest file, we determined the full set of implicit Intent types that the application's components were willing to accept, and from the Java source we determined the set of Intent types that the application could potentially send. Then, by taking the intersection of these two sets, we were able to determine what kinds of implicit Intents could be sent from one component of the application to another.

The static analysis tool was able to successfully decompile and analyze 487 of the top 504 applications. [2] 2.15% of the Intents in this sample were flagged by the tool as implicit Intents used for intra-application communication. In 115 of the sampled applications, this type of Intent appeared at least once. To evaluate the correctness of our static analysis tool, we picked a random sample of 20 applications and manually reviewed them to determine the rate of false positives and the percentage of reported true positives that were actual security vulnerabilities. In this sample, a total of 65 Intents were flagged as implicit Intents used for intra-application communication; three were false positives. Of the remaining 62 identified true positives, 60 could lead to Intents being delivered to the wrong recipient.

B. Automatic Exporting of Components

One dangerous consequence of using implicit Intents for internal communication is that components listen for them by declaring an Intent filter, which will automatically export them unless they specify otherwise. Therefore, we investigated how often components are exported this way and how often our modifications would prevent them from being exported. To answer these questions, we changed the `PackageParser` code for Android, which is responsible for parsing the manifest file that is packaged with each application. Of the 497 applications we tested, 344 would be exported in a normal Android system but not in our modified version of Android.

From the 344 applications that have components which are automatically exported but which would not be exported with our changes, we selected a random sample of 20 for detailed investigation. We found that 14 of these 20 applications were vulnerable to simple denial-of-service attacks: by sending simple but carefully constructed Intents from an unprivileged

[1] We had planned to analyze the top 504 but were unable to complete seven of the downloads.

[2] The `jad` decompiler was not able to decompile ten of the applications in the downloaded 497.

process we were able to crash them. Most of the exploits involved inducing null pointer exceptions by omitting parts of the Intent that the receiving component expected. We crashed five applications by omitting the Intent action string, four by omitting an expected extra field, one by omitting the Intent's data URI, and one by setting an invalid extra field. For the remaining three vulnerable applications the cause of the crash was unclear.

Four of the 20 applications appeared to expose functionality that was intended for internal use only and could potentially be abused. The most worrisome of these was PayPal's free Android application, which allows a user to send electronic payments to others. We found that by sending appropriate Intents, we could construct a payment of an arbitrary amount, payable to an arbitrary party, while bypassing all but one of the normal interaction screens. The single screen we failed to avoid was the final "Send" confirmation screen; the user must still tap that button to complete the transaction. However, we came up with several possible schemes for tricking a user into doing so. For example, the malicious application could masquerade as a game that encouraged the victim to tap rapidly and repeatedly in the area of the "Send" button; while that was going on a carefully timed Intent could be sent to the PayPal application, causing it to bring up its confirmation screen just as the user tapped.

V. Defense Evaluation

Having verified that the vulnerabilities we identified are both real and dangerous, and having implemented a defense, we set out to evaluate the practicality of our approach.

A. Effectiveness Against Existing Exploits

As mentioned above, we found 62 cases where implicit Intents were used for internal communication in the 20 applications we analyzed manually. Of those 62, our defense prevents 60 of them by delivering the Intents to components in the same application that sent them.

Of the 14 Intent spoofing vulnerabilities we found, our defense eliminates 10 of them by preventing the vulnerable components from being automatically exported by Android.

B. Effect on Existing Applications

We did not find any issues regarding the Intent interception defense and its compatibility with current applications on Google Play. We manually reviewed 20 randomly selected applications to test the effect of our defense, including applications with and without Intents affected by our defense to observe its effects on both kinds of apps. Our testing did not uncover any cases where applications were broken by the Intent interception defense.

However, we found that our Intent spoofing defense breaks some functionality in all 20 of the applications we examined by hand. Most of the problems involved one or more of the following: Android's billing system, the Cloud to Device Messaging framework (C2DM), the app widget system, Amazon in-application purchasing, live wallpapers, analytics, ad networks, and Broadcast Receivers such as those that listen for SMS and MMS messages.

Fortunately, many of these problems can be easily resolved by making two changes to the criteria suggested by Chin et al. [1] to decide whether to block a component from being exported. First, if a component is protected by a permission, we should allow it to be exported. This would solve the problems we observed with C2DM, Amazon in-application purchasing, and live wallpaper, because the components that used these features correctly used the permissions system to restrict which applications could send Intents to them. This change would not sacrifice much security, because components that are already protected by permissions have little need for additional protection.

A second change, which would resolve most remaining backward compatibility problems, would be to allow a component to be automatically exported unless it filters exclusively for Intents within its own namespace. Usually when a component filters for Intents outside of its application's namespace, it does so because it expects to receive Intents from other applications. By allowing these components to be exported, we can ensure that they can receive the Intents they need for correct functionality.

VI. Related Work

Introductions to Android security can be found in Burns [2] and in Enck et al. [3]. Relevant application characteristics are surveyed by Enck et al. [4] and Felt et al. [5], [6].

Chin et al. [1] focused on how to secure applications from each other by improving the safety of inter-application communication, particularly communication via Intents. They identified two broad classes of Intent vulnerabilities, Intent spoofing and unauthorized Intent receipt, and developed a tool, ComDroid, which they used to disassemble and statically analyze DVM bytecode. This approach uncovered many vulnerabilities in a sample of existing Android applications. Based on their results, they presented recommendations for changes to how Intents are handled; these recommendations form the basis for (but are not identical to) our own modifications to Android.

One potential consequence of Intent vulnerabilities is privilege escalation: by sending a cleverly crafted Intent, a malicious application might escalate its own permissions by tricking a vulnerable application into doing its bidding with higher permissions. Davi et al. present a proof-of-concept that uses a (now fixed) vulnerability in the Phone application to mount a privilege escalation attack [7]. Felt et al. consider "confused deputy" and "unintentional deputy" attacks [8] and present IPC Inspection, which ensures that an application cannot enhance its privileges by delegating a task.

SCanDroid [9] is a tool that tracks data flows through and across application components, identifying flows that violate the security policy implied by the relevant permissions. Kirin [10] is a mechanism that enforces rules about what sets of permissions are permissible to applications. The user blacklists certain combinations of permissions (for example,

the ability to record sound and send it over the network) to prevent applications from gaining dangerous capabilities. TaintDroid [11] uses dynamic taint analysis to monitor the flow of sensitive data on the phone and determine when it is leaving the phone (e.g. as an SMS message). Ongtang et al. [12] present Saint, a system for enforcing application security policies.

None of these approaches address the vulnerabilities in Intent handling that we consider, though SCanDroid might be modified to add such protections.

VII. Conclusion and Future Work

We have shown that Android's system of handling inter- and intra-application messages (Intents) has vulnerabilities that allow malicious applications to take advantage of innocent ones. An extensive study of applications available on Google Play showed that the threat is not theoretical, and in at least one case could allow a malicious developer to steal money from unsuspecting users.

In addition to the changes suggested in Section V-B, we suggest a modification to the way Android's Intent filtering mechanism behaves. A common misunderstanding arises with Intent filters, because declaring a filter does not actually "filter out" any Intents. Paradoxically, declaring a filter for a component greatly widens the set of Intents that it can receive, because (unless it is protected by a permission) Android's default is to open up the component to receive any explicit Intent from any application. Our evidence suggests that developers incorrectly use Intent filters to guarantee certain input preconditions, causing invalid behavior when these preconditions are not met. We recommend changing Android's Intent delivery so that it enforces the criteria of Intent filters on explicit Intents as it does for implicit ones, at least for communication between separate applications.

Although we have addressed the most common varieties of Intent vulnerabilities, Android's Intent system has a handful of subtle corners that could use more attention in the future. Sticky broadcasts persist after they have been sent and cannot be protected by permissions. Ordered broadcasts can be halted or tampered with by malicious applications. Another interesting issue is that Intents can delegate authority either by using pending intents or by sending Content Provider URI's.

Finally, there are also subtleties with respect to dynamically registered Broadcast Receivers. Unless a permission is specified, they are open to receive Intents from any application on the phone. Furthermore, prior to 4.0, Android ignores `Intent.setPackage()` when it is resolving a broadcast to a dynamically registered Receiver, suggesting that it may often be possible to intercept broadcasts simply by registering a Receiver dynamically. All of these issues could use more research into how prevalent they are in the wild and how to mitigate related vulnerabilities.

Trademarks

References

[1] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252.

[2] J. Burns, "Mobile application security on Android," ser. Blackhat'09, 2009.

[3] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *Security Privacy, IEEE*, vol. 7, no. 1, pp. 50 –57, jan.-feb. 2009.

[4] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21.

[5] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14.

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638.

[7] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proceedings of the 13th international conference on Information security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346–360.

[8] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22.

[9] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automatic security certification of Android applications," University of Maryland, College Park, Tech. Rep. CS-TR-4991, November 2009.

[10] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235–245. [Online]. Available: http://doi.acm.org.ezproxy.libraries.claremont.edu/10.1145/1653662.1653691

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[12] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," in *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, dec. 2009, pp. 340 –349.