

Exploiting common **Intent** vulnerabilities in Android applications

Kelly Casteel, `kcasteel`
Owen Derby, `oderby`
Dennis Wilson, `dennisw`

December 12, 2012

Problem

The Android framework allows apps and components within apps to communicate with one another by passing messages, called Intents, which effectively specify both a procedure to call and the arguments to use. Applications must declare in a static manifest file which Intents each component services, as well as both application and component level permissions. While the security vulnerabilities in outgoing Intents have been well studied [1] and developer tools exist to limit potentially insecure Intents [5], little has been done to address malicious incoming Intents.

Exploits of this nature have been discovered in firmware of various Android phones [4], but exploits in third-party applications are not well studied. Application developers must make sure their manifest file has been properly configured to only accept desired Intents, which can limit usability. We believe that developers will trust Intent input by default, allowing malicious input to potentially crash or abuse the application.

Our work is twofold: we developed a static analysis tool to inspect third party applications for malicious Intent vulnerabilities, and we built a working exploit which takes advantage of such a vulnerability.

Static analysis

To start, we implemented a basic static analysis tool to aid us in identifying potential vulnerabilities in Android applications. We are not the first to look at such vulnerabilities [1], nor to build such a tool [4]. However, in the absence of any freely-available such tools, we took it as a learning opportunity to build our own, borrowing ideas freely from the previous literature. Here, we describe the basic design of our tool and any interesting decisions we made, but the curious reader should reference [1] and [4] for more details.

We built our tool using Androguard [2], a python FOS library built, in part, to support the creation of static analysis tools for the Android platform. It supports disassembly and decompilation of apks, Android application packages, into Java-approximate source code for human-readability, as well as intermediate basic blocks for static analysis and control-flow graph creation. It provides basic search functionality over the decompiled basic blocks. Despite this long list of features, a lot of work went into understanding how to use the library and coercing it into doing what we wanted.

Our tool examines execution traces through an application, looking for places where malicious Intents might cause the program to perform unintended privileged actions. We implement the component analysis strategy from [1] (see section 4.3) to detect open components. We also identify all privileged calls made by the application [3]. Then we use a control flow graph of the program to determine which open components lead to privileged calls - these paths represent potential vulnerabilities.

First, we extract the manifest file from the apk and compile a list of “open” components: public components, protected by weak or no permissions which are exported by the app. A component is public if the manifest specifies an Intent filter for it. However, developers can explicitly make components private (regardless of any intent filters) by setting the “exported” attribute to `false` for each component in the manifest file. Developers can set the “permission” attribute to require a certain permission to access each component, thereby restricting access to the component. We only considered public, exported components which had permissions of level `dangerous` or `normal`.

After determining all open components, we need to identify all privileged calls made by the application. A privileged call is a method call (usually Android API call) which the Android OS requires the calling application to hold a permission in order to execute. The mapping from API calls to required permissions is derived from work in [3]. Using this mapping and the decompiled source code in the apk, we search for privileged calls with permission levels of `dangerous` or `signature`.

Finally, we construct a control flow graph of the application and search for any paths of execution leading from an open component to a privileged call. We ignore paths between open components and privilege calls which require the exact same permission. We report any such path found as a potential vulnerability in the analyzed application. These paths represent possible vulnerabilities because they allow an unprivileged application to cause execution of privileged methods via an Intent.

Because our intention was to quickly discover many vulnerabilities in 3rd party applications, but not exhaustively so, we did not attempt to handle discontinuities in the control flow graph resulting from callbacks (see 2.1.1 in [4]). This means there are possibly many more vulnerabilities in the applications we examined than what we reported. Further, because many non-harmful methods still require permissions, there are a lot of “vulnerabilities” identified which are not very harmful in practice. For example, `getNumberFromIntent` requires `dangerous` permission `CALL_PRIVILEGED`. However, without any additional vulnerabilities in the application, invoking such a privileged call via intent is not very interesting. Many of the applications flagged by our tool had this sort of “harmless” vulnerability.

Gathering apps for analysis

We downloaded 382 applications as apk files using Real Apk Leecher [6], a utility we found online. This connects to Google Play to get applications, so all results were verified applications in the Android market. Various keywords for potential interesting attack vectors, such as “camera,” “contacts,” “sms,” and more were used to search for applications.

The free applications from the top 60 search results for each keyword, sorted by application popularity, were downloaded for analysis. This ensured that the applications were popular and currently used applications; the attacker could be fairly certain that some were already downloaded by a potential victim.

Table 1: Permission use and leakage

Permission	Use	Vulnerabilites
INTERNET	319	97
READ_PHONE_STATE	183	60
ACCESS_FINE_LOCATION	140	36
READ_LOGS	51	14
READ_CONTACTS	120	11
CHANGE_COMPONENT_ENABLED_STATE	10	7
GET_TASKS	47	6
CAMERA	68	5
DISABLE_KEYGUARD	23	5
AUTHENTICATE_ACCOUNTS	17	3
CALL_PRIVILEGED	7	3
SEND_SMS	32	3
BLUETOOTH	29	2
NFC	7	1
READ_SMS	45	1
Total		254

Android permissions leaked in the analyzed applicatons. Use indicates the number of applications that declared the permission in their manifest, and vulnerabilities indicates the number of applications that exposed one or more vulnerabilities involving permission.

Permissions that were declared but not exposed are not shown. Signature or system permissions are in bold, all others are dangerous.

Findings of analysis

The permissions most commonly leaked were the permissions most used by applications. Either INTERNET, READ_PHONE_STATE, or ACCESS_FINE_LOCATION were leaked in 113 of 382 applications. While exploits are possible with these permissions, the more active permissions (and thus provide more interesting exploits if leaked), such as CAMERA or SEND_SMS, were much less frequently exposed. The correlation between exposed permissions and common ones suggests that developers don't consider the security risks of the common permissions. Users are also probably more willing to simply allow an app to have these permissions.

We were surprised to discover that none of the flagged vulnerabilities represented leaks of sensitive or privileged information. We confirmed this by checking to see if any of the apps made calls to setResult. Very few did, and most were only statically setting result codes (i.e. result codes were not informative of whether the activity request succeeded or not).

We looked at apps which called any of the Android API calls to extract data from an Intent, like getBundleExtra, getCharArrayExtra, getIntExtra, etc, and found that many were properly sanitizing the input, but some were not. An example is Vault Hide SMS, Pics & Videos, an app that allows users to store SMS messages, contacts, call logs,

photos, and videos in a password-protected and encrypted space. This app leaks the permissions `CHANGE_COMPONENT_ENABLED_STATE`, `GET_TASKS`, `INTERNET`, `READ_PHONE_STATE`, and `SEND_SMS`. The message sender gets number information from the user and message data from an Intent, allowing an attacker to send arbitrary text in place of the user's original message. While the user still enters the number and knows that the text message is being sent, this could potentially be used to generate SMS spam.

Close inspection of the decompiled application code returned by our application led to the discovery of similar potential attack vectors. Some are not linked to any one permission, but rather the functionality of the application, and are therefore not in the current scope of the static analysis tool. Detection of these vulnerabilities from decompiled bytecode proved difficult and may be more suited for source code analysis.

The decompiled application code for active permissions such as `CAMERA` or `SEND_SMS` revealed that most applications require user confirmation or alert the user of the permission use inherently by changing screens or displaying input text before execution. An example of this is the `CAMERA` permission; applications including the system camera app allowed Intents to start the camera. However, once the camera is started, the user will see the camera output and will have to click for a picture to be taken. Android does provide a system level permission `INJECT_EVENTS` that lets the application inject events such as clicks into the event stream for any window, but this permission is purposefully warned against by Android and would be suspicious in any app. The attack vector is further limited by the transfer of control to the vulnerable application; the malicious application can not regain control after sending an Intent in most applications.

Internet Based Exploits

The most commonly leaked permission was `INTERNET`, and of the 97 applications returned by our static analysis tool as exposing that permission, 7 allowed arbitrary URLs to be loaded in the app's WebView. For example, we found two apps, Jazz Internet Radio and Sky.FM, both of which import `CatalogActivity` from the Flurry library, which will load any URL sent to it in an Intent. Although a malicious application which exploited these vulnerabilities not receive the response from the request, the application could submit forms containing any data that the it had access to as well as visit malicious websites.

This has several security related implications. First, it allows the attacker to exfiltrate data from the app that the user believed would be used only by the application. Users may be less careful about giving sensitive data to apps that they believe do not have access to the internet. Also, a malicious app could connect to an ad server and generate fraudulent ad clicks, while potentially depleting the user's data plan. Finally, the ability to force the user to visit an attacker-controlled URL means that attackers can learn the IP address of any user, which is one of the factors used to determine the coarse location of the device, which is supposed to be protected.

Interestingly, the stock Android browser will also load any URL that it receives in an Intent. This means that any app can force the browser to visit any link using the user's cookies; it is reasonable to assume that a mobile user would visit many cookie-saving sites in the default browser. Phishing attacks can therefore be embedded in apps that

were not supposed to have access to the internet in the first place.

Example Exploit

We built an exploit to demonstrate the vulnerability we found in the Android browser. We developed an app that appears to the users to be a private diary. It does not request any permissions. However, when the user tries to save the diary entry, the app URL encodes the entry and uses the browser to submit it to a website we control. If the user has a login cookie saved for the zoobar website, our website changes the user's zoobar profile to the diary entry along with a timestamp and the user's IP address. Control is then returned to the application using a link click on our website, which creates an Intent. Our application's Intent filter catches Intents of that specific URI scheme (`malware://`) and returns control to the application.

Conclusion

Our analysis of 3rd party apps looked for possible ways a malicious Intent could trigger privileged calls by the app. We found that over half of the apps examined leaked one or more privileged calls. However, very few (7 out of 97, for INTERNET-leaking apps) leaked interesting, actionable exploits.

While our static analysis tool checked for permission differences between entry and exit points in applications, we only found permission escalation from no permissions to those enumerated above. This suggests that app developers are not checking the permissions of incoming Intents, otherwise we would have seen a greater diversity of entry point permissions. A search for questions about the `checkCallingPermissions` methods on StackOverflow, the now official Android help forum, does not return many results which supports the hypothesis that developers are not using this feature. The default Android applications also allow privilege escalation through Intents; we found privilege escalation paths in both the default browser and camera apps. However, there are very few applications that rely on Intent data for important or sensitive parts of their functionality.

The static analysis tool we developed does not completely detect permission escalation, and future work could be done to refine the tool. However, this would mostly support potential attackers in finding exploitable applications. The tools [5] for protecting against privilege escalation exploits already exist, along with default Android functions like `checkCallingPermissions` that limit incoming Intents. Developers do not seem to be using these tools, which suggests that the Android platform needs better vetting of applications before allowing them in the app market.

References

- [1] 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, page 239252, New York, NY, USA, 2011. ACM.
- [2] Anthony Desnos. Androguard. <https://code.google.com/p/androguard/>.

- [3] 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*, page 627638, 2011.
- [4] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.
- [5] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security, CCS '12*, page 229240, New York, NY, USA, 2012. ACM.
- [6] Wu Dang Thang. Real apk leecher. <http://forum.xda-developers.com/showthread.php?t=1563894>.