Sleeping Android: The Danger of Dormant Permissions

James Sellwood
Royal Holloway University of London
Egham, United Kingdom
james.sellwood.2010@live.rhul.ac.uk

Jason Crampton
Royal Holloway University of London
Egham, United Kingdom
jason.crampton@rhul.ac.uk

ABSTRACT

An Android app must be authorized for permissions, defined by the Android platform, in order to access certain capabilities of an Android device. An app developer specifies which permissions an app will require and these permissions must be authorized by the user of the device when the app is installed. Permissions, and the tools that are used to manage them, form the basis of the Android permission architecture, which is an essential part of the access control services provided by the Android platform.

We have analyzed the evolution of the Android permission architecture across six versions of the Android platform, identifying various changes which have occurred during that period and a considerable amount of information about the permission architecture which is not included in the Android documentation. Using this information, we have identified a weakness in the way that the Android platform handles app permissions during platform upgrades. We explain how this weakness may be exploited by a developer to produce malicious software which the average user is unlikely to detect. We conclude with a discussion of potential mitigation techniques for this weakness, highlighting concerns drawn from other research in this area.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement

General Terms

Security

Keywords

Android; permissions; permission architecture; authorization; privacy; malware

SPSM'13, November 8, 2013, Berlin, Germany.

ACM 978-1-4503-2491-5/13/11 ...$15.00.
http://dx.doi.org/10.1145/2516760.2516774.

1. INTRODUCTION

Mobile operating systems such as the Android platform employ various security mechanisms in order to manage operation of the mobile device. The mechanisms are employed to balance the interaction of the user, their device, the installed apps and the associated data. There are three principal security mechanisms employed by the Android platform with varying levels of rigour and effectiveness: application sandboxing, application permissions and installation controls.

Application sandboxing limits the resources with which an app can interact, and in so doing it prevents malicious apps from affecting the behaviour and data of other apps and the underlying Android platform. Application permissions provide a mechanism through which an app may be authorized to make use of particular capabilities of the Android platform or of other apps. Informally, application sandboxing is a straight-jacket and the permissions are the bindings which keep it tight or allow it to be loosened. Finally, installation controls are present to restrict the mechanism through which apps are installed. The Android platform can be configured to only allow apps to be installed from the official Google Play app store, or alternatively can allow third-party sources to be used. When restricted to the official app store only, these controls, along with the Google Play policies, help provide some measure of confidence in the provenance of an app. When third-party sources are enabled this measure of confidence is removed and it is the user's sole responsibility to determine the trustworthiness of any app they choose to install.

This paper focuses on the application permissions architecture. There has previously been significant research into the Android permission architecture, with a wide variety of approaches so far undertaken. We consider some of this related work in Section 6. Our contribution to this research is taken from two distinct viewpoints.

Firstly, we investigated the categorisations employed within the permissions architecture. Permissions are organised both in functional groupings, within which similar permissions are collated, and in terms of protection levels, which dictate the requirements which must be met for a permission to be granted. By setting such requirements, protection levels indicate a degree of sensitivity for the permissions categorised under them. Functional groupings are equally designed so as to provide the user information about the permissions, with the hope of aiding the authorization decisions a user must make when installing apps.

Secondly, we investigated how these categorisations have changed over time, as new versions of the Android platform have been introduced. This view of the history and development of the Android platform is of particular interest when attempting to identify the rationale for the current permission architecture configuration. Android versions are associated with an API level and a codename. Table 1 summarizes Android versions since January 2010; bold entries indicate API levels used in our research. Henceforth, we will refer to Android versions by API level.

Version	**Codename**	**API Level**
2.1	Eclair	7
2.2.x	Froyo	**8**
2.3–2.3.2	Gingerbread	9
2.3.3–2.3.7		**10**
3.0	Honeycomb	11
3.1		12
3.2.x		**13**
4.0.1–4.0.2	Ice Cream Sandwich	**14**
4.0.3–4.0.4		**15**
4.1.x	Jelly Bean	**16**
4.2.x		17
4.3		18

Table 1: Android versions, codenames and API levels

Thirdly, and most significantly, we used the results of these investigations to identify a hypothetical weakness in the Android permission architecture. Informally, the weakness arises because permissions are able to lie dormant and only awake following an update to the Android platform, either because a new app is installed or because Android itself is upgraded. A dormant permission is one that is undefined when an app (that requests the permission) is installed. Undefined permissions are ignored at installation time. The permission may become active if it is subsequently defined, either as a third-party permission in another app that is installed by the user or as a built-in permission in a later version of Android. The user is normally required to authorize an app to use permissions during the installation of an app. However, the activation of a dormant permission occurs without any notification to the user or any authorization by the user.

Further experiment proved this weakness to be present and easy to exploit. We document it here so as to raise awareness of the potential for malicious exploit of the permissions architecture and to prime the discussion of potential mitigation strategies.

In Sections 2 and 3 we describe our two permission architecture investigations and their results. Then in Section 4 we discuss third-party permission granting and its relevance to the results of our investigation . In Section 5 we describe the threat identified by these investigations, its exploitation, the repercussions and potential mitigation. We discuss related work in Section 6 and provide concluding remarks in Section 7.

2. PERMISSION ARCHITECTURE

An Android app typically needs to interact with the device on which it is installed. The Android platform enables such interactions by associating permissions with an app. When a user installs an app on their Android device they may be prompted to authorize certain permissions which it requests: "Each application must declare upfront what permissions it requires" [20]. The developer documents the required permissions in the app's `AndroidManifest.xml` file and the Android platform analyzes this list, displaying its contents to the user, during installation.

Each permission is classified under one of a variety of protection levels. For example, those permissions which control access to the lowest risk capabilities are considered `normal` whilst those of higher risk are considered `dangerous`. During installation, those permissions which are rated `dangerous` will be displayed to the user for their authorization whilst those rated `normal` will be presented in a collapsed list which must first be expanded before the individual permission requests can be seen. The authorization process is atomic such that "users may only grant all requested permissions or deny them all by not installing the application" [10]. Once installed, the permissions for which an app is authorized can be reviewed via the app's 'app info' screen, as seen for one of our test apps in Figure 2. Granted permissions are listed in two sections, `dangerous` followed by `normal`, with the `normal` section collapsed by default. Functional groupings are present within these two sections bringing together permissions which control access to similar features.

There are a significant number of permissions built into the Android platform which control access to various capabilities of the device. For example, location-based services are controlled by `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` whilst use of the device's internet connection is controlled by `INTERNET`. Android also allows developers to define their own ("third-party" defined) permissions. Such permissions are not concerned with any underlying Android system capabilities but are instead used to control inter-app communication mechanisms such as Intents [8]. Developers defining third-party permissions can dictate the protection level associated with that permission. Moreover, there are several other protection levels available in addition to the `normal` and `dangerous` protection levels already mentioned: `signature` requires the requesting app to be signed with the same key as the app which defined that permission; whilst `signatureOrSystem` extends `signature` by allowing requesting apps to be part of the system image.

There are many scenarios when the `signature` protection level is likely to be most appropriate for third-party permissions. When used to protect inter-app communication, permissions with this protection level will limit that communication to only those apps signed with the same key. That said, there are equally legitimate scenarios which would require other protection level settings and the developer is able to define the setting necessary for their particular circumstances, functional requirements and risks.

As well as a protection level, every permission has associated with it a label and a functional grouping. The label provides a short user-oriented definition, such as 'approximate (network-based) location' for the permission `ACCESS_COARSE_LOCATION`, whilst the grouping is used to collate similar permissions together. So, for example, all those permissions associated with a user's location information are grouped under the heading 'Your location'. The protection level, label and functional grouping make up the core at-

Permission Test Manifest

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2     package="com.escapadesinsecurity.android.permission.test"
3     android:versionCode="1"
4     android:versionName="1.0" >
5 
6     <!-- control permission - known `dangerous' permission -->
7     <uses-permission android:name="android.permission.INTERNET" />
8     <!-- All 130 permissions -->
9     <uses-permission android:name="android.permission.ACCESS_CHECKIN_PROPERTIES" />
10    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
11    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
12    <uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
13    <uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
14    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
15    <uses-permission android:name="android.permission.ACCESS_SURFACE_FLINGER" />
16    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
17    <uses-permission android:name="android.permission.ACCOUNT_MANAGER" />
18    <uses-permission android:name="com.android.voicemail.permission.ADD_VOICEMAIL" />
19    <uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
20    <uses-permission android:name="android.permission.BATTERY_STATS" />
21    <uses-permission android:name="android.permission.BIND_ACCESSIBILITY_SERVICE" />
22    <uses-permission android:name="android.permission.BIND_APPWIDGET" />
23    <uses-permission android:name="android.permission.BIND_DEVICE_ADMIN" />
24    <uses-permission android:name="android.permission.BIND_INPUT_METHOD" />
25    <uses-permission android:name="android.permission.BIND_REMOTEVIEWS" />
26    <uses-permission android:name="android.permission.BIND_TEXT_SERVICE" />
27    <uses-permission android:name="android.permission.BIND_VPN_SERVICE" />
28    <uses-permission android:name="android.permission.BIND_WALLPAPER" />
29    <uses-permission android:name="android.permission.BLUETOOTH" />
30    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
31    <uses-permission android:name="android.permission.BRICK" />
32    <uses-permission android:name="android.permission.BROADCAST_PACKAGE_REMOVED" />
33    <uses-permission android:name="android.permission.BROADCAST_SMS" />
34    <uses-permission android:name="android.permission.BROADCAST_STICKY" />
35    <uses-permission android:name="android.permission.BROADCAST_WAP_PUSH" />
36    <uses-permission android:name="android.permission.CALL_PHONE" />
37    <uses-permission android:name="android.permission.CALL_PRIVILEGED" />
38    <uses-permission android:name="android.permission.CAMERA" />
39    <uses-permission android:name="android.permission.CHANGE_COMPONENT_ENABLED_STATE" />
40    <uses-permission android:name="android.permission.CHANGE_CONFIGURATION" />
41    <uses-permission android:name="android.permission.CHANGE_NETWORK_STATE" />
42    <uses-permission android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE" />
43    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
44    <uses-permission android:name="android.permission.CLEAR_APP_CACHE" />
45    <uses-permission android:name="android.permission.CLEAR_APP_USER_DATA" />
46    <uses-permission android:name="android.permission.CONTROL_LOCATION_UPDATES" />
47    <uses-permission android:name="android.permission.DELETE_CACHE_FILES" />
48    <uses-permission android:name="android.permission.DELETE_PACKAGES" />
49    <uses-permission android:name="android.permission.DEVICE_POWER" />
50    <uses-permission android:name="android.permission.DIAGNOSTIC" />
51    <uses-permission android:name="android.permission.DISABLE_KEYGUARD" />
52    <uses-permission android:name="android.permission.DUMP" />
53    <uses-permission android:name="android.permission.EXPAND_STATUS_BAR" />
54    <uses-permission android:name="android.permission.FACTORY_TEST" />
55    <uses-permission android:name="android.permission.FLASHLIGHT" />
56    <uses-permission android:name="android.permission.FORCE_BACK" />
57    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
58    <uses-permission android:name="android.permission.GET_PACKAGE_SIZE" />
```

Figure 1: *Permission Test* `AndroidManifest.xml` (lines 1 to 58)

tributes of the permission and form the basis upon which a user is expected to make their authorization decisions when installing an app.

2.1 Initial Investigation

The Android documentation web site currently lists 130 platform permissions [5]; this list being used as the starting point of our investigation. We created a test app, called *Permission Test*, which included each of these 130 permissions in its manifest (see Figure 1), although the app's sole function was to display a 'Hello World' message on the device's screen. As the manifest permission requests are known to trigger the authorization process even without an associated need within the app's code base [20, 10, 11], this simplification avoided the complex, time-consuming and potentially error-prone process of developing an app that exercised each of the capabilities associated with those permissions.

The *Permission Test* app was installed on a stock Galaxy Nexus device running API15 and which of the requested list of 130 manifest permissions was granted was determined by review of the app info screen seen in Figure 2.Where necessary we also made use of internal Android system files, in particular the `data/system/packages.xml`, the Android logging system, and the testing of individual permissions.

Access to the `data/system/packages.xml` is not possible on a stock device without elevated privileges. In order to achieve the desired access without impacting the investigation's results we manually 'rooted' the device, when necessary, using minimal, reversible steps and without modifying the operating system image. In this way we were able to extract the data required whilst ensuring accurate and repeatable results.

2.2 Initial Investigation Findings

Our results, as broken down in Table 2, show that 79 of the 130 manifest permissions were granted to *Permission Test*. Log entries tagged 'PackageManager' helped us identify the fate of the other 51 permissions.

The Package Manager is the Android service responsible for installation and app management. Three warning types were seen and are documented below.

- 4 instances of 'Unknown permission [...] in package com.[...].permission.test'
- 24 instances of 'Not granting permission [...] to package com.[...].permission.test (protectionLevel=2 flags=0x8be46)'
- 23 instances of 'Not granting permission [...] to package com.[...].permission.test (protectionLevel=3 flags=0x8be46)'

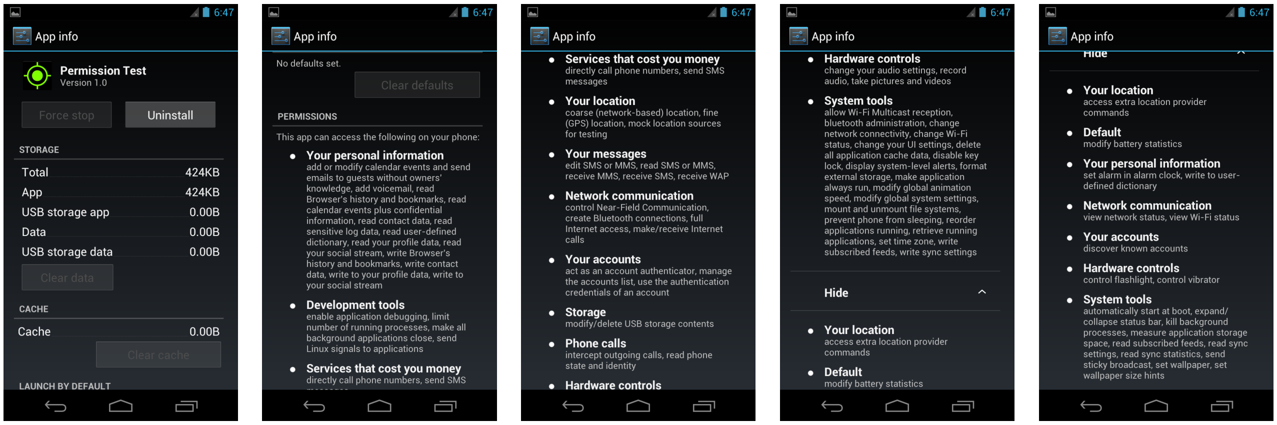
A comparison with the Android documentation [5, 7] revealed that the four manifest permissions associated with the 'Unknown permission' warning were introduced in API16 and so were 'unknown' to the version of the Android platform running on the device (API15). The documentation also confirmed that 'protectionLevel=2' indicates the `signature` protection level whilst 'protectionLevel=3' indicates the `signatureOrSystem` protection level. Therefore, whilst the log file did not contain detailed information about labels or groupings, it did provide protection level information for 47 of the 51 manifest permissions which hadn't been granted. The classification of these 47 permissions also confirmed why they had not been granted to *Permission Test*: the app was not signed by the necessary key nor was it part of the system and so failed to meet the criteria necessary to have them granted.

Installing *Permission Test* on the Android emulator running API16 resulted in the test app being granted three of the four unknown permissions, two of which were classified as `dangerous` and the other was classified as `normal`. The one remaining permission was not granted, but examination of the system log identified a 'PackageManager' entry showing the fourth permission was a `signature` protection level permission.

3. PERMISSION EVOLUTION

It is worth noting at this point that whilst we identified four permissions which were attributed the 'Unknown permission' warning on API15, the Android documentation actually identifies six permissions as having been "Added in API Level 16" [3]. It was these unknown permissions and this discrepancy with the documentation which highlighted the need for more understanding of the evolution of the permission architecture. This discrepancy will be discussed further in Section 3.3.

In order to catalogue the permissions and their attributes on various Android platform versions we employed six instances of the Android emulator, one for each of the API versions highlighted in Table 1 (APIs 8, 10, 13, 14, 15 and 16).



(a) (b) (c) (d) (e)

Figure 2: *Permission Test* app information on API15

Group	`dangerous` Permissions	`normal` Permissions	Total Permissions
Your personal information	13	2	15
Development tools	4	0	4
Services that cost you money	2	0	2
Your location	3	1	4
Your messages	5	0	5
Network communication	4	2	6
Your accounts	3	1	4
Storage	1	0	1
Hardware controls	3	2	5
Phone calls	2	0	2
System tools	19	11	30
Default	0	1	1
-	59	20	79

Table 2: API15 permission groupings

Having collected information about both those permissions granted and not granted, we saw that each of the attributes (protection level, permission label and functional grouping) showed some degree of change as the Android platform evolved. As well as the permissions' attributes, some permissions' very existence underwent change, as had been suggested by our initial investigation. Space restrictions prevent us form reporting all our findings, which are described in a technical report available on-line [24].

3.1 Protection Level Definitions

The analysis of the Android system logs on the emulator running API16 showed a considerable change in the protection level assignments when compared with previous API versions. On API versions prior to API16 there were four protection levels, as previously indicated. These four protection levels were assigned four numerical values: 0 for `normal`, 1 for `dangerous`, 2 for `signature` and 3 for `signatureOrSystem`. Whilst these four protection levels exist in API16, two new flags were introduced and assigned hex values of `0x10` for the `system` flag and `0x20` for the `development` flag [7].

These flags allow for a far greater combination of values than just the previous four, although in the analysis we only saw two new combinations used. The first had a hex value of `0x12` indicating a protection level of 2 with the new `system` flag set. This is interpreted as `signature` or `system` and seemed to overlap with the already existing `signatureOrSystem` protection level. The second had a hex value of `0x32` indicating a protection level of 2 with both the `system` and `development` flags set. This protection level is interpreted as `signature` or `system` or `development` and had no previous equivalent. The original protection levels, the new flags and the two new combinations are shown in Table 3.

The assignments to the new `0x12` protection level on API16 suggested that this is a direct replacement for the existing `signatureOrSystem` protection level. Where previously there had been as many as 23 permissions classified as `signatureOrSystem`, in API16 this dropped to zero and 21 permissions were classified under the new `0x12` protection level. Every one of these 21 permissions was previously classified as `signatureOrSystem` in API15.

In API16 there were eight permissions classified under the new `0x32` protection level. These eight permissions were previously classified under several of the other protection levels — two were `signatureOrSystem` whilst six were `dangerous`.

	Decimal	Hex	Binary	Protection Level
Before API16	0	0x00	0000 0000	`normal`
	1	0x01	0000 0001	`dangerous`
	2	0x02	0000 0010	`signature`
	3	0x03	0000 0011	`signatureOrSystem`
API16 Flags	-	0x10	0001 0000	`system`
	-	0x20	0010 0000	`development`
API16 Combinations	18	0x12	0001 0010	`signature` or `system`
	50	0x32	0011 0010	`signature` or `system` or `development`

Table 3: Protection Levels

3.2 Permission Re-classification

The re-classifications associated with the new protection levels in API16 were just part of the permission re-classifications identified during the cataloging of permissions. Across the entire catalogue there were 36 re-classifications with 21 of these associated with changes from `signatureOrSystem` to `0x12`, as identified above, and therefore considered less significant from a security point of view. Every one of the 36 re-classifications involved a move to a new classification which was at least as restrictive as the old classification. The 15 changes which were not associated with the change from `signatureOrSystem` to `0x12`, occur across the evolution of 14 of the 130 permissions, as shown in Table 4.

Of these 14 permissions, there are several worthy of mention. The `DISABLE_KEYGUARD` permission enables an app to 'disable keylock' on the device. This permission started out as a `normal` permission and was re-classified in API10 to `dangerous`. It does seem strange that such a permission was originally considered to be so innocuous, but hindsight is often biased by experience. Another interesting permission is `DUMP`. This permission started out as `dangerous` due to the fact, as its label describes, it allows an app to 'retrieve sensitive internal state'. In API10 it was re-classified to `signatureOrSystem` and it was re-classified again in API16 to `0x50` (`signature` or `systems` or `development`). This latest move does not further restrict use of the permission, but does hint at making the capability more useful to developers, who likely would have been unable to make easy use of it whilst it was classified as `signatureOrSystem`.

3.3 Permission Growth

Whilst certain permissions have been re-classified during the evolution of the Android permission architecture, new permissions have been added with most new versions of the platform. As shown in Table 5 there have been as many as seven new permissions added from one version to the next. From our baseline version of API8 there were three additions in API10, two additions by API13, seven additions in API14, no additions in API15 and four additions in API16. In total 16 permissions were added between API8 and API16, approximately 12% of the 130 permissions documented in API16. Whilst the classification of permissions to protection levels changed across the platform versions as already discussed, most new permissions were assigned to the `signatureOrSystem` (or its equivalent `0x12`) and `0x32` protection levels.

As we saw at the start of Section 3, it was the four unknown permissions we first identified in API15, when compared against the Android documentation's list of 130 permissions for API16 [5], which triggered this investigation. The Android documentation, however, identifies six permissions as being new to API16 [3] but only four of these were not granted to us when using API15. In fact the two supposedly new permissions which were granted to us, `READ_USER_DICTIONARY` and `WRITE_USER_DICTIONARY` were granted during installation of *Permission Test* on each of the emulated platforms, going as far back as API8. A check of the Android source code also showed that these permissions existed in API8, as our results indicated. It is unclear at this time why these are documented as new to API16 based on this evidence. It is clear from the results so far discussed and from the Android source code that a large number of permission related changes were performed between API15 and API16. This does seem to include changes to the groupings which these two permissions are assigned to, but they are just a small part of a much wider change and seem to be the only two permissions which are incorrectly highlighted as new.

Interestingly, there were two other discrepancies of this nature. The Android documentation identifies five permissions as being added in API14 and two in API15 [1, 2]. Our experiments suggest that all seven permissions were added in API14, with none added in API15.

3.4 Permission Re-labeling

As we have seen, in particular from the results shown in Table 5, there were a large number of changes made in API16. In addition to those already mentioned, a further significant change was that 47 permissions underwent a re-labeling between API15 and API16. A few permissions had previously been re-labeled, but as part of API16's now obvious widespread permission changes, approximately 36% of permissions were re-labeled. The re-labeling looks to have been performed so as to make it clearer to users which capabilities a permission enables an app to use. Some examples of this include:

- `ACCESS_COARSE_LOCATION` — 'coarse (network-based) location' was changed to 'approximate (network-based) location'
- `AUTHENTICATE_ACCOUNTS` — 'act as an account authenticator' was changed to 'create accounts and set passwords'
- `SYSTEM_ALERT_WINDOW` — 'display system level alerts' was changed to 'draw over other apps'

3.5 Permission Group Change

Finally, we identified one change to the functional group assignment of a permission during the analysis. The

	Protection Level (decimal)					
Permission	**API8**	**API10**	**API13**	**API14**	**API15**	**API16**
CHANGE_COMPONENT_ENABLED_STATE	2	3	3	3	3	18
DISABLE_KEYGUARD	0	1	1	1	1	1
DUMP	1	3	3	3	3	50
MODIFY_PHONE_STATE	1	3	3	3	3	18
READ_FRAME_BUFFER	2	2	2	3	3	18
READ_LOGS	1	1	1	1	1	50
SET_ALWAYS_FINISH	1	1	1	1	1	50
SET_ANIMATION_SCALE	1	1	1	1	1	50
SET_DEBUG_APP	1	1	1	1	1	50
SET_PROCESS_LIMIT	1	1	1	1	1	50
SIGNAL_PERSISTENT_PROCESSES	1	1	1	1	1	50
UPDATE_DEVICE_STATS	2	3	3	3	3	18
WRITE_APN_SETTINGS	1	1	1	3	3	18
WRITE_SECURE_SETTINGS	3	3	3	3	3	50

Table 4: Protection Level changes across platform versions

	Permission Count					
Number of ...	**API8**	**API10**	**API13**	**API14**	**API15**	**API16**
`normal` (0) permissions	20	20	20	20	20	21
`dangerous` (1) permissions	54	55	55	59	59	55
`signature` (2) permissions	24	22	23	24	24	25
`signatureOrSystem` (3) permissions	16	20	21	23	23	0
`0x12` (18) permissions	0	0	0	0	0	21
`0x32` (50) permissions	0	0	0	0	0	8
Unknown permissions	16	13	11	4	4	0
New permissions	-	3	2	7	0	4
Re-labeled permission	-	2	0	2	0	47

Table 5: Android permissions breakdown across platform versions

READ_LOGS permission was moved from the ‘System tools’ group in API8 to the ‘Your personal information’ group in API10.

3.6 New Versions of Android

Two new versions of the Android platform have been released since our original analysis: API17 was released at the end of 2012 and API18 was released mid 2013. We briefly summarize some of the main changes.

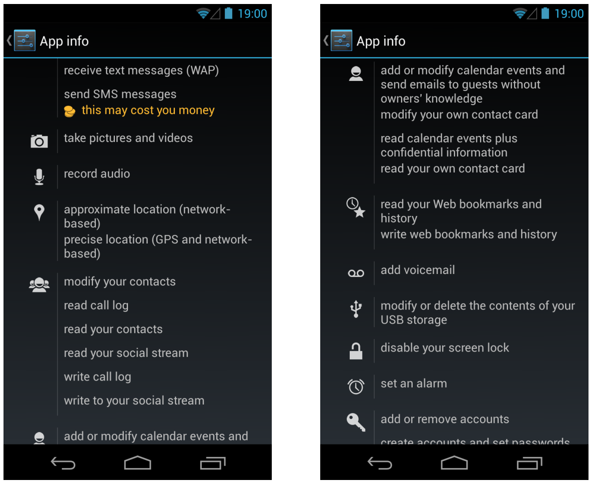
The Android documentation lists 130 permissions [5] for API16. However, our analysis of the Android source code revealed the API16 release code contained 180 permissions and 12 functional groups. As of API17, the Android documentation continues to list 130 permissions [5] and, when compared to API16, no permission additions or changes are indicated [4]. Our analysis of the API17 release code, however, suggests that there have been a variety of modifications. These modifications include the addition of 20 permissions, taking the total in the Android source code to 200 permissions and 31 functional groups. Of these 20 permissions, two are duplicates of existing permissions and seem to have been added to the source by mistake.

Of those permissions that already existed, five permissions moved to a more restrictive protection level although, more significantly from a security standpoint, eight permissions moved from `dangerous` to `normal`. This reduction in protection level was not seen across any of the previous platform updates that we have investigated. These results can be seen in Table 6.

	Protection Level (decimal)	
Permission	**API16**	**API17**
BATTERY_STATS	0	1
CHANGE_CONFIGURATION	1	50
CHANGE_NETWORK_STATE	1	0
MODIFY_AUDIO_SETTINGS	1	0
MOUNT_FORMAT_FILESYSTEMS	1	18
MOUNT_UNMOUNT_FILESYSTEMS	1	18
PERSISTENT_ACTIVITY	1	0
REORDER_TASKS	1	0
SERIAL_PORT	0	18
SET_TIME_ZONE	1	0
WAKE_LOCK	1	0
WRITE_SETTINGS	1	0
WRITE_SYNC_SETTINGS	1	0

Table 6: Protection Level changes in API17

As well as these changes to the permissions themselves, API17 modifies the app info screen and how permissions are displayed to users (compare Figure 3 to Figure 2). Permissions of different protection levels are no longer separated, with the `normal` permissions included alongside the `dangerous` ones and all permanently visible. Whilst the changes to



(a) (b)

Figure 3: *Permission Test* app information on API17

the app info screen may make the display more consistent, the user can no longer distinguish, and therefore no longer draw benefit from the classification of, `normal` and `dangerous` permissions. In fact, the changes to the app info screen may obscure changes in `dangerous` permissions as there are now more permissions to examine, compared to the default display in API16.

The release of API18 included an update to the number of documented permissions, from 130 to 134 [5]. A preliminary analysis of the source code indicates that in fact 9 new permissions have been added whilst three have been removed. Of those three, two of them are the duplicates added in API17.

As with API17, API18 also makes a permission architecture related modification apart from the permission additions and deletions just mentioned. In the case of API18, a new, yet currently hidden, interface has been identified which seems to identify the permissions an app is coded to make use of, rather than just relying on which permissions it requests in its `AndroidManfiest.xml`. This 'App ops' screen suggests that in some future version of Android the user will be able to selectively grant or deny permissions.

4. THIRD-PARTY PERMISSIONS

Having catalogued changes to the Android permission architecture across a number of platform versions, it was obvious that the Platform Manager was processing permission requests by comparing the manifest permissions requested by an app against some reference list within the platform. This reference list was obviously changing over time and it was the existence of a permission on this list, along with the protection level assigned to it, which determined whether a permission would be unknown, granted or denied when requested by an app at installation time.

4.1 Investigation

One final aspect of the permission architecture was left to investigate, that of third-party defined permissions. Recall that requesting a built-in permission that was not available on a given platform would result in an 'Unknown permission' log entry (see Section 2.2). It seemed likely that requesting a third-party permission which was not yet defined would result in a similar log entry. In order to test this hypothesis we created two new test apps, *Permission Test Creator* and *Permission Test Requestor*.

Permission Test Creator defines a third-party permission `com.escapadesinsecurity.android.permission.value.TEST` and categorises it under the `dangerous` protection level. It also requests this permission and the built-in `INTERNET` permission, used here as a control. *Permission Test Requestor* on the other hand did not define any third-party permissions, but did request the same two permissions, `com.escapadesinsecurity.android.permission.value.TEST` and `INTERNET`.

In order to determine if our expectations were correct, we installed the two test apps one after the other on the Galaxy Nexus device. The first time we installed *Permission Test Creator* before *Permission Test Requestor*, reviewing the granted permissions as we went. Here we were requesting only permissions that had been defined. We then uninstalled both apps from the device. The second time we installed *Permission Test Requestor* before *Permission Test Creator*, once again reviewing the granted permissions as we went. Here we were requesting a third-party permission before it was defined.

4.2 Findings

4.2.1 *Creator then Requestor*

Having first installed *Permission Test Creator*, its app info screen was reviewed and both the `com.escapadesinsecurity.android.permission.value.TEST` and `INTERNET` permissions were listed as granted. *Permission Test Requestor* was then installed and its app info screen was reviewed. This also showed both permissions as granted.

This series of events aligns with the operation of built-in permissions. When the permission being requested exists, in this case because the defining app is already installed, then the permission is granted.

4.2.2 *Requestor then Creator*

Having uninstalled both apps, we first installed *Permission Test Requestor*. Its app info screen was then reviewed and only the `INTERNET` permission was listed as granted. *Permission Test Creator* was then installed and its app info screen reviewed. Both the `com.escapadesinsecurity.android.permission.value.TEST` and `INTERNET` permissions were listed as granted. This is what we expected: when the requesting app was installed prior to the permission being defined, it was unable to be granted; whereas *Permission Test Creator* was obviously able to request the permission it itself defined, just as it had been during the first test.

A subsequent review of the *Permission Test Requestor* app info screen, however, revealed that the `com.escapadesinsecurity.android.permission.value.TEST` permission had been added to the previously listed `INTERNET` permission. This suggested that the permissions were being re-evaluated in some way, even though the requesting app was not modified after initial installation.

It was this observation combined with the knowledge gained from the evolution investigation which highlighted an obvious question — If third-party permissions, which are unknown prior to the defining app being installed, are in some way re-evaluated post definition, does the same thing happen with unknown built-in permissions?

4.3 Further Analysis

Before we move on to discuss the exploit hypothesis which naturally flows from the question just raised, we discuss a number of other findings applicable to third-party permissions revealed by our investigation.

4.3.1 App Info Screen

During the investigation just described, the two test apps were written in the same vein as the original *Permission Test*. Neither actually made use of the third-party defined permission, they simply requested it in their manifest as the 130 built-in permissions had been before. During further analysis, new versions of these two apps were created with the third-party permission employed to control Intent access to the *Permission Test Creator* app. *Permission Test Requestor* was coded to make use of an Intent, and through it to pass a text string for display by *Permission Test Creator*. If the third-party defined permission was not requested by, or granted to, *Permission Test Requestor* then it would be unable to trigger *Permission Test Creator*.

We ran the two experiments as described in Section 4.2, but this time, as well as reviewing the app info screens, the Intent function was triggered within *Permission Test Requestor* as a test of the ability to use the permission. Unexpectedly, we found a discrepancy in the second experiment (where *Permission Test Creator* is installed after *Permission Test Requestor*). Even though the `com.escapadesinsecurity.android.permission.value.TEST` permission was listed on the *Permission Test Requestor* app info screen once *Permission Test Creator* was installed, an 'Access Denial' exception was thrown when the Intent was triggered.

A review of the `data/system/packages.xml` system file confirmed that the permission had not been granted. It would seem that whilst the app info screen had been updated, once the permission had been defined, the actual permission requested had not been granted to the app. Considering this screen is a user's sole indicator of an installed app's permissions, such inaccuracies further reduce a user's abilities to make security and privacy decisions about the apps that they install.

4.3.2 Change in Protection Level

Having identified that a permission categorised as `dangerous` is not actually granted when defined after the installation of a requesting app, we modified *Permission Test Creator* so the protection level of the permission was `signature`. In this case the permission was granted and the Intent was successful (even though the defining app was installed after the requesting app). The difference this time was that the permission was not listed on the app info screen (although this was to be expected as it was now categorised as `signature`).

These observations further highlight the complexity of the Android permission architecture and some of the inconsistencies in its implementation on the Android platform.

5. DORMANT REQUESTS

To summarize: we have discovered that if (i) we install an app whose manifest includes a third-party permission that is unknown to the platform and (ii) that permission is subsequently defined by a newly installed app, then that "dormant" permission will become available to the first app. This raises the following important question: Does the same thing happen with unknown built-in permissions?

5.1 Hypothesis

We used this question as the basis for the following hypothesis: Could a malicious app be created that includes built-in permissions which are not defined in the version of the platform on which the app is initially installed, but which "awaken" at a later time, without the user's knowledge, when the device's platform is upgraded?

Three requirements would have to be met for such a malicious app to be possible. Firstly, a developer would need to be able to identify permissions which are to be introduced in future versions of the Android platform, and then build an app which did not crash or otherwise indicate to a user its intent whilst running on the older platform. This requirement is met, since the Android platform, has changed over time and, no doubt, will continue to do so. Google already announce and document changes in the platform so that developers can write software making use of the latest features [6]. This documentation is often in place well before the majority of users have even heard of the new platform version, let alone have hold of a device which has it installed. It is also trivial for a developer to have their app identify the version of the platform it is running on, using a call to `Build.VERSION.SDK_INT` for example, and then modify the actions taken based on the result.

Secondly, new permissions would need to be introduced and be configured to protect sensitive capabilities within the Android platform. Without this, whilst a developer could technically write an app to request unknown permissions, they wouldn't have any permissions to request that they could take malicious benefit from. This requirement is known to already have been met. We have already identified the existence of such unknown permissions and in fact of those introduced in API16, two of these, `READ_CALL_LOG` and `WRITE_CALL_LOG`, are marked with a protection level of `dangerous` and likely to be considered by many as protecting sensitive information. API16 also introduced the `READ_EXTERNAL_STORAGE` permission which sounds perfect for malicious activity. It was, however, introduced without mandatory enforcement within the platform. Instead it was introduced as a precursor to future platform restriction and was initially only present to aid adoption by developers.

Thirdly, the question posed at the start of this section must have a positive answer. There must be some re-evaluation present for built-in permissions in the same way that we had witnessed for third-party permissions. In order to determine if this final requirement was also met, one further investigation was required.

5.2 Malicious App

In order to test our hypothesis we wrote a new test app, *Permission Test Jelly Bean*. This app was coded to request three permissions within its manifest — `READ_CALL_LOG`, `WRITE_CALL_LOG` and `READ_EXTERNAL_STORAGE`. These were three permissions already identified as being new to API16.

The test app first identifies the platform version on which it has been installed. If it finds it isn't on API16 then it performs no action, other than writing the following message to the system log: 'ABC123: I'm not on Jelly Bean, I'm on SDK??, so I must be good - shhh, don't draw attention to yourself!' (where SDK?? is replaced with the actual SDK version the app is running on). If the app instead identifies it is running on API16 then it performs a series of operations:

1. The app writes the message 'ABC123: I'm on Jelly Bean, so I can be bad - shhh, don't draw attention to yourself!' to the system log.
2. The app uses the `READ_CALL_LOG` permission to access the user's call log.
3. The app writes the phone numbers found in the call log to the system log.
4. The app uses the `WRITE_CALL_LOG` permission to write a new entry to the user's call log.
5. The app documents this new entry in the system log.
6. The app verifies whether external storage is present in the device.
7. The app uses the `READ_EXTERNAL_STORAGE` permission to read the folder and files in the root of the external storage (if present).
8. The app writes the names of any folders or files found into the system log.

No matter whether running on API16 or not, the test app always displays the same 'Hello World' message to the user via the device screen. In this way the app has a visually consistent behaviour on all platform versions.

It should be clear that *Permission Test Jelly Bean* fits the requirements for a malicious app attempting to make use of unknown permissions. The app makes use of the permissions only on API16, and uses them whilst providing no visual indication to the user (other than that entailed in demonstrating use of the permissions in question, such as writing a false call log entry).

5.3 Exploit

The app was installed on a Galaxy Nexus device running API15. During the installation there was no prompt for permissions to be authorized by the user. A review of the app info screen showed no `dangerous` or `normal` permissions as having been granted.

The test app was run, and, as coded, the only visual indication was the 'Hello World' message displayed on the screen. When the system log was reviewed a single log entry was present as shown in Figure 4a.

A system update was then applied, using the standard Android process, and the platform updated to API16. The update process ends with the device booting into the newly installed API16 version of the Android platform. Once the device restarted, the test app was re-run and once again the 'Hello World' message was displayed. This time however, when the system log was reviewed it was clear that malicious action had been taken by the app. Multiple log entries were present, as shown in Figure 4b. It had read four entries from the call log and wrote an additional random entry back to it. The app also found 11 directories (d) within the root of the external storage and four files (f).

At no time after the test app's installation were we prompted to authorize the permissions it requested; not even when the system update was performed, those permissions were defined and the device restarted. At the time *Permission Test Jelly Bean* was run on API16 there had been no visual cue to the user that the permissions had changed. A review of the app info screen did now show the permissions as granted however. This change had, just like the re-evaluation of third-party permissions, completed automatically in the background when the permissions were defined and without providing any notification to the user.[1]

5.4 Repercussions

Our successful test had shown that it is indeed possible to request built-in permissions prior to their introduction in a newer version of the Android platform. We also showed that an app that requested these permissions could disguise this fact and then maliciously utilise these permissions once a device was upgraded to the platform version which defined them. Whilst the test app we developed only performs local logging actions, it is not difficult to envisage a more profitable scenario for a malicious developer.

For example, a malicious app could be constructed as a weather app, and could legitimately request the `INTERNET` and `ACCESS_COARSE_LOCATION` permissions in order to retrieve weather information for the user's location (two permissions commonly requested by weather apps and widgets [18]). These two permissions would allow the app access to network connectivity and approximate location information and could be used by the app, whilst running benignly, to provide weather information. During the installation it is unlikely that many users would reject such permission requests. Should the user update their device platform, which for many would simply entail clicking an install button when they are prompted that an over-the-air system update is available, then the malicious app's dormant permission requests would become active. The app could then use those permissions to access sensitive information, assuming applicable new permissions, and could then misuse the two legitimate permissions to send that information along with the user's location over the Internet to the app's developer.

It would only be through reviewing the app's app info screen after the update, and thus viewing the currently granted permissions, that the user would have any chance of becoming aware of the new capabilities available to the app. Whilst this review would enable the user to determine the discrepancy and uninstall the app, the likelihood that users regularly perform those checks at this time is, we believe, extremely low. We also believe that many users would fail to identify the new permissions as such and would most likely assume they had been present all the time, therefore being less likely to take any corrective action. Whilst research has been performed into user understanding and awareness of permissions [19, 23], at this time we are not aware of any research being performed into user's recollection of what permissions an app has (without reviewing) nor what an app should have (from a blank slate). These are appropriate ar-

[1]The exploit works perfectly well with APIs 17 and 18 (although the dormant permissions in API(x) relative to API($x + 1$) differ for each x).

Tag	Text
PTJB	ABC123: I'm not on Jelly Bean, I'm on SDK15, so I must be good - shhh, don't draw attention to yoursel f!

(a) System log entry on API15

Tag	Text
PTJB	ABC123: I'm on Jelly Bean, so I can be bad - shhh, don't draw attention to yourself!
PTJB	ABC123: Found 4 records in the call log. - shhh, don't draw attention to yourself!
PTJB	ABC123: Phone Entry 0: 02070313000 (2012-08-03 22:15:01.635) - shhh, don't draw attention to yourself!
PTJB	ABC123: Phone Entry 1: +447782333123 (2012-08-03 22:14:39.215) - shhh, don't draw attention to yoursel f!
PTJB	ABC123: Phone Entry 2: 07734265938 (2012-08-03 22:14:22.075) - shhh, don't draw attention to yourself!
PTJB	ABC123: Phone Entry 3: 07734265938 (2012-08-03 22:14:03.787) - shhh, don't draw attention to yourself!
PTJB	ABC123: Wrote new entry for 414243313233830 to the call log. - shhh, don't draw attention to yourself!
PTJB	ABC123: File 0: Music (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 1: Podcasts (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 2: Ringtones (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 3: Alarms (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 4: Notifications (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 5: Pictures (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 6: Movies (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 7: Download (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 8: DCIM (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 9: Android (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 10: .estrongs (d) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 11: Permission Test.apk (f) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 12: Permission Test Creator.apk (f) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 13: Permission Test Requestor.apk (f) - shhh, don't draw attention to yourself!
PTJB	ABC123: File 14: Permission Test Jelly Bean.apk (f) - shhh, don't draw attention to yourself!

(b) System log entries on API16

Figure 4: *Permission Test Jelly Bean* log entries

eas for future research considering the weakness identified here.

Hopefully our test app has shown that to achieve such a scenario would not be difficult. We required no complex development or reverse engineering expertise in order to achieve our goals. We simply wrote a standard Android app which requested some permissions and made use of them on a specific version of the Android platform. This is exactly what many Android developers likely do with every app they write.

5.5 Potential Mitigation

The ultimate cause of the weakness we identified is the re-evaluation of permissions at the time of system update. It could be argued however, that in reality the actual security threat associated with the weakness stems from this re-evaluation being performed automatically in the background, with no indication of any change being presented to the user. In this way, the user is unable to make the security decisions about an app which they are otherwise required to make.

It would be easy to suggest that permissions should not be automatically re-evaluated and granted without user interaction. This would certainly align with the original definition of the `dangerous` protection level, which is, as stated by Google [7]:

> "A higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system may not automatically grant it to the requesting application. For example, any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities."

There are, however, reasons why such a suggestion may be short-sighted. Much research has been performed in the area of permission authorization and user interaction for security decisions [19, 23, 21, 22, 17, 13]. The general consensus is that users are not particularly good at making security-related decisions, although the reasons for this vary. In some cases there is seen to be a distinct lack of user knowledge which could be considered a pre-requisite for such decisions. In others, the users do not perceive the risk as significant enough to decide against installing the app. In some other cases, the users simply do not see the responsibility to be theirs. There is also anecdotal evidence from users themselves regarding their willingness to participate in such decisions, with 58% of 5,950 Android Central forum users admitting "I just click right through" [15].

Adding another permission authorization decision may not therefore be the answer. If users do "just click right through" then this is likely just another dialogue they will ignore. An alternative to asking the user to explicitly authorize the changes identified by re-evaluation would be to simply indicate to the user that a change has occurred and provide the means to review and revoke these permissions should they desire. For example, the Android notification system [9] could be employed, where the notification also includes actions made available to review the permissions of the app or to uninstall the app. Whilst this would not meet Google's definition of the `dangerous` protection level, it would at least provide awareness and give users the ability to make an informed decision. At this time, even this is lacking and so the user is left unduly exposed.

These two potential mitigations are no doubt not the only options and, based on the breadth of research in this area, it is clear that the problem of user-device interaction in the context of security decisions is one that is both complex and as yet unresolved. We believe one thing is however clear, the current automatic and invisible re-evaluation of permissions leaves users unable to make the very decisions the Android platform demands of them. This flaw in the implementation of the permission architecture requires resolution.

6. RELATED WORK

Our work considers the way in which Android permissions have evolved over time, how permissions are referenced by apps at installation time, and how the interaction of these two aspects of Android permission management combine to create the threat of dormant permissions. We have also examined ways in which the threat posed by dormant permissions might be reduced, concluding that the attitude of users to authorizing permissions is central to the efficacy of any mitigations. We are unaware of any work in the literature that covers similar ground. However, we briefly discuss the most closely related concepts and research below.

One might think of the problem of dormant permissions as a new type of time-of-check-to-time-of-use (TOCTTOU) problem. Such a problem "occurs when a program checks for a particular characteristic of an object, and then takes some action that assumes the characteristic still holds when in fact it does not" [14]. TOCTTOU flaws can, for example, lead to access to sensitive files or to continued access to protected resources even after authorization has been revoked. Dormant permissions mean that capabilities not available to an app become available at some point in the future, because permission authorization is only checked when the app is installed, not when the set of permissions changes (due to the installation of a third-party app or a platform upgrade).

Au *et al.* examined the permission architectures of various smartphone platforms in general, and Android in particular [10]. They highlighted the problem of apps having more privileges than required and also tracked the number of permissions added in early versions of Android (up to 2.3). Porter Felt *et al.* modified the permission verification mechanism of Android 2.2 in order to identify which API calls result in permission checks. They then built 'Stowaway', a tool for static analysis of apps, and used it to determine permission errors (particularly overprivilege) within the apps tested [20]. Vidas *et al.* also analysed apps to identify permission errors, highlighting duplicate permissions and overprivilege in particular [26]. Barrera *et al.* developed a framework for evaluating permission-based security models and used their methodology to analyze the Android permission architecture [12], which led to suggestions for improving the structure of the Android permission architecture. While all the work described above is concerned with Android permissions, none of this prior research considers the way in which the evolution of the permission architecture might affect the permissions that are installed by apps.

Chia *et al.* considered the permissions used by apps for a number of platforms, including Android [16]. They focused on comparing the permissions requested by popular and newly added apps as well as considering the effectiveness of risk signals available to users. Shin *et al.* identify a means of abusing third-party permission requests through the fact that permission strings can be easily copied and are the only identifying requirement for uniqueness [25]. This work considers some of the risks posed by permissions used by third-party apps, but does not link this with the issue of dormant permissions and permission evolution.

There have been several user studies designed to evaluate various aspects of users' understanding of Android permissions and the implications of making particular choices with regard to permissions [17, 19, 22, 23]. Porter Felt *et al.* have also investigated a number of alternative mechanisms by which users can interact with permissions and suggested that different permissions required different methods of interaction [21]. This work will certainly be useful in determining the best way of mitigating the risk posed by dormant permissions, something we hope to consider in future work.

7. CONCLUSION

We have undertaken an analysis of the Android permission architecture and its history. In doing so we have found that numerous changes have occurred in the core attributes associated with permissions — protection levels, functional groups, and labels. It would seem that the changes so far implemented have all been made with a view to increasing the security of the platform, either through the restriction of a permission or through an attempt to improve the user's understanding of what a permission allows. We have not performed any analysis as to how successful such changes have been and we believe an interesting piece of further research would be to determine whether users' comprehension of permissions has been increased through the changes so far made.

Through our analysis we have identified the outcome of requesting various types of permissions, built-in permissions, unknown permissions, third-party defined permissions and undefined third-party permissions. The platform response to these allowed us to predict the presence of a weakness in the permission architecture which we have proven to be present. We have developed a simple test app which exploits this weakness and have detailed a scenario under which a developer could produce an app to exploit the weakness for malicious gain.

We concluded by describing two potential mitigations to the weakness and have highlighted some primary discussion points regarding which mitigation is preferable. In future work, we hope to develop user studies to determine which mitigations are likely to be most effective.

8. REFERENCES

[1] ANDROID. API Differences between 13 and 14. `http://developer.android.com/sdk/api_diff/14/changes.html`. [online] Accessed On: 2013-05-19.

[2] ANDROID. API Differences between 14 and 15. `http://developer.android.com/sdk/api_diff/15/changes.html`. [online] Accessed On: 2013-05-19.

[3] ANDROID. API Differences between 15 and 16. `http://developer.android.com/sdk/api_diff/16/changes.html`. [online] Accessed On: 2013-05-19.

[4] ANDROID. API Differences between 16 and 17. `http://developer.android.com/sdk/api_diff/17/changes.html`. [online] Accessed On: 2013-06-20.

[5] Android. Manifest.permission | Android Developers. http://developer.android.com/reference/android/Manifest.permission.html. [online] Accessed On: 2013-08-22.

[6] Android. Platforms | Android Developers. http://developer.android.com/tools/revisions/platforms.html. [online] Accessed On: 2013-05-19.

[7] Android. R.attr | Android Developers. http://developer.android.com/reference/android/R.attr.html#protectionLevel. [online] Accessed On: 2013-05-19.

[8] Android. Security Tips (IPC) | Android Developers. http://developer.android.com/training/articles/security-tips.html#IPC. [online] Accessed On: 2013-06-13.

[9] Android. Status Notifications | Android Developers. http://developer.android.com/guide/topics/ui/notifiers/notifications.html. [online] Accessed On: 2013-05-19.

[10] Au, K. W. Y., Zhou, Y. F., Huang, Z., Gill, P., and Lie, D. Short Paper: A Look at SmartPhone Permission Models. In *Proceedings of the First ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011), SPSM'11, ACM.

[11] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (2012), CCS'12, ACM.

[12] Barrera, D., van Oorschot, P., Kayacik, H., and Somayaji, A. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the Seventeenth ACM Conference on Computer and Communications Security* (2010), CCS'10, ACM.

[13] Benenson, Z., Hintz, N., Kroll-Peters, O., and Krupp, M. Poster: Attitudes to IT-Security When Using a Smartphone. In *Eighth Symposium on Usable Privacy and Security (SOUPS)* (2012), SOUPS'12.

[14] Bishop, M., and Dilger, M. Checking for race conditions in file accesses. *Computing systems 2*, 2 (1996), 131–152.

[15] Central, A. Late-Night Poll: Do You Read App Permissions Before Installing? http://www.androidcentral.com/late-night-poll-do-you-read-app-permissions-installing. [online] Accessed On: 2013-05-19.

[16] Chia, P. H., Yamamoto, Y., and Asokan, N. Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *Proceedings of the Twenty-First International World Wide Web Conference* (2012), WWW'12, ACM.

[17] Egelman, S., Porter Felt, A., and Wagner, D. Choice Architecture and Smartphone Privacy: There's A Price for That. In *Proceedings of the 11th Annual Workshop on the Economics of Information Security* (2012), WEIS'12.

[18] Google. weather - Google Play. https://play.google.com/store/search?q=weather&c=apps. [online] Accessed On: 2013-05-19.

[19] Kelley, P. G., Consolvo, S., Cranor, L. F., Jung, J., Sadeh, N., and Wetherall, D. A Conundrum of Permissions: Installing Applications on an Android Smartphone. In *Proceedings of the Workshop on Usable Security* (2012), USEC'12.

[20] Porter Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android Permissions Demystified. In *Proceedings of the Eighteenth ACM Conference on Computer and Communications Security* (2011), CCS'11, ACM.

[21] Porter Felt, A., Egelman, S., Finifter, M., Akhawe, D., and Wagner, D. How to Ask for Permission. In *Proceedings of USENIX Workshop on Hot Topics in Security (HotSec) 2012* (2012).

[22] Porter Felt, A., Egelman, S., and Wagner, D. I've Got 99 Problems, But Vibration Ain't One: A Survery of Smartphone Users' Concerns. Tech. rep., University of California, Berkeley, 2012.

[23] Porter Felt, A., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android Permissions: User Attention, Comprehension, and Behavior. In *Eighth Symposium on Usable Privacy and Security (SOUPS)* (2012), SOUPS'12.

[24] Sellwood, J. Sleeping Android: Exploit Through Dormant Permission Requests. Tech. Rep. RHUL–MA–2013–6, Royal Holloway, University of London, 2013. Available from http://www.ma.rhul.ac.uk/static/techrep/2013/MA-2013-06.pdf.

[25] Shin, W., Kwak, S., Kiyomoto, S., Fukushima, K., and Tanaka, T. A Small but Non-negligible Flaw in the Android Permission Scheme. In *Proceedings of the 2010 IEEE International Symposium on Policies for Distributed Systems and Networks* (2010), POLICY'10, IEEE.

[26] Vidas, T., Christin, N., and Cranor, L. F. Curbing Android Permission Creep. In *Proceedings of the Web 2.0 Security and Privacy 2011* (2011), W2SP'11.