Sleeping Android: the danger of dormant permissions

Changes to the way the Android platform authorises permission requests could compromise the security of unwary users

by James Sellwood and Jason Crampton
Since the first commercial device was made available in October 2008, the Android platform has enjoyed a meteoric rise. In those four years it has grown to hold the greatest market share among many of the world’s most significant smartphone markets. Competing head to head with Apple’s iOS platform, the two operating systems are used in the vast majority of the world’s smartphones.

The Android platform has evolved considerably since its introduction. Since 2008, there have been 25 platform version releases – the latest being 4.2.1 – introducing 17 different API levels in that time. These releases have introduced numerous new features and, as one might expect nowadays, they have also included various bug fixes and security patches.

One area of the Android platform has undergone continued development in that time and has received close scrutiny due to its significant security role and noticeable disparity from similar mechanisms on other platforms. The area in question is the Android permission architecture.

Permission architecture basics

Android’s permissions are used by developers to request access to the many capabilities made available by the platform. Without requesting the appropriate permission, an app is unable to make use of the relevant capability. Well-known examples include the “internet” permission, which allows network access by an app, and the “Access_fine_location” permission, which allows precise location determination (using GPS, for example). Permissions are authorised by the user during app installation, an activity that many Android users skip through with little understanding or attention.

To make permissions easier to manage and understand, they are organised by group and protection level. Permission groups are used to organise permissions based on similar functions with examples such as “Services that cost you money”, “Your accounts” and “Your messages”. These groups and their structure are designed to make it easier for a user to understand the purpose of permissions and have been a part of the permission architecture from the beginning. Recently, with the release of Android 4.2, the representation of permissions within their groups has been made more prominent during app installation.

Protection levels, on the other hand, indicate the potential risk associated with permissions. They provide guidance to the developer and the end-user as to the level of concern that should be associated with the granting of a particular permission, and thus making the associated capabilities available to an app.

The base protection level is “normal”, with permissions in this level being considered sufficiently low risk that, by default, they are not displayed to the user during authorisation. (Instead, they are hidden under a collapsed display, which must be expanded by the user for them to be observed.)

The next protection level is “dangerous”, which contains permissions considered higher risk. Permissions in the dangerous protection level must be displayed to...
the user for authorisation and are permissions that expose the user, their data or their money.

Both the normal and dangerous permissions are available to any app once authorised. Permissions associated with the higher protection levels are accessible only to those apps that are signed under specific code-signing keys. The next protection level is “signature”. To request permissions in this level, an app must be signed under the same key as the app that defined the permission. In the case of signature permissions defined by the Android platform, everyday developer apps are unable to gain these permissions and the user will not be prompted to authorise any requests made by such apps. The final protection level is “signatureOrSystem”, which works in a similar way to the signature level with the addition that apps installed in the system image can be granted these permissions. Once again, permissions defined by the Android platform and assigned to this protection level are inaccessible to everyday developers.

While iOS developers and users have a handful of permissions to deal with, the Android permission architecture contains hundreds. As of Jelly Bean (API 17), the relevant page in the Android documentation listed 130 permissions. However, thorough inspection of the source code suggests there are in fact around 200 Android permissions. Of those 200 permissions, 29 have a protection level of normal, 47 are dangerous, 62 are signature and 62 are signatureOrSystem. That means that the everyday developer can request up to 76 permissions, selected on the basis of the capabilities required by the app. In reality though, developers on average require three or four permissions to achieve the goals of their apps.

Permission architecture evolution

The permission architecture, like other parts of the Android platform, has undergone many revisions with both additions and modifications. It is worth pointing out that the general rationale for these changes has been to enhance
the security of the platform. While it is clear that the Android platform is not, and has never been, perfect, the changes made to the permissions architecture have led to more control, greater privacy and less data exposure.

Two of the most obvious examples of this are changes made between API8 and API10. Originally a permission called “Disable_keyguard” was available as a normal protection level permission. As its name suggests, this permission allows an app to disable the keyguard and thus unlock the Android device. This permission was changed from a normal permission – one that by default would not be displayed to the user during installation – to dangerous – one that would be displayed to the user. The “Dump” permission, which allows an app access to significant service state information, was available as a dangerous permission in API8. By API10, this permission had been upgraded to a signature protection level and was thereby no longer accessible to everyday developers.

It is worth noting that not all protection level changes occurred during the early versions of the platform. The “Write_apn_settings” permission, for example, allows an app to configure a device’s mobile access point address, which could enable an attacker to reroute data traffic. This permission was dangerous protection level until API13 and was then increased to signature protection level at API14.

While there are a number of interesting and important protection level changes that have occurred across the version releases, there are other changes, which, at first sight, seem to have less significant security ramifications. The label associated with a permission is the short text string, which is displayed on the authorisation screen during app installation. There have been many permission label changes which, while not directly affecting security controls, do modify the user experience. Such changes have the potential to affect security understanding and therefore user behaviour. For example, the “Authenticate_accounts” permission originally had the label “act as an account authenticator”. This was changed during a significant relabelling that occurred in API16 to “create accounts and set passwords”, which is a far more informative and noteworthy description of what an app can do given that permission. The relabelling performed in API16 was so extensive that 47 of the 76 normal and dangerous permissions underwent changes.

When evolution exposes weakness

The examples listed above are just some of the many findings made during my investigation into the Android permission architecture and its evolution. My most significant discovery concerned the mechanism the platform uses to authorise an app’s permission requests and the nuances of the interactions that occur when the platform is updated on a device.

When an app is installed, the platform identifies the permissions requested by parsing the relevant entries in the app’s manifest file (AndroidManifest.xml). These permission requests are then compared with the platform’s list of defined permissions and associated protection levels. The user is then prompted to authorise all of the normal and dangerous permissions the app requested and that were found to be defined by the platform API. Any undefined permissions are ignored and any permissions of higher protection levels are handled based on the rules applicable to that level. The user must authorise all displayed permissions or decline the installation. Once installed, the app will be able to make use of any capabilities defined by the permissions for which the app is authorised.

The alert reader may have noticed there are several ways in which individual permission requests may not be authorised and yet the app can still be installed. Permissions associated with protection levels of signature or signatureOrSystem
but where the app does not meet the necessary requirements is one such example. Of more interest here, however, is the fact that permissions that are not recognised by the platform cannot be authorised and the user will not be made aware of such permission requests during installation of the app.

This second point has particular relevance when we consider what happens when the device platform is updated and specifically when the update includes the addition of new permissions. When the platform update completes, the device reboots and the platform re-evaluates the permissions associated with the apps that had previously been installed on the device. This process does not involve user interaction: all normal and dangerous permissions are authorised automatically in the background. Presumably, the assumption is that the app was already installed, so these permissions had already been authorised by the user. As we have seen, this may not be the case because a previously undefined permission – a “dormant” permission – may have since been defined in the more recent platform version.

These specific interactions result in a weakness in the permission architecture that can be exploited by an attacker to create malicious software. Moreover, the technique requires nothing more than the awareness of a new permission that can be misused. Given that Android announces the details associated with new Android platforms well in advance of them being made available to the general public, awareness is not a problem. And history has shown that almost any technology, in the wrong hands, can, with some imagination, be misused or abused.

To illustrate, let us consider the example of a malicious weather app. This app could legitimately request the “Internet” and “Access_coarse_location” permissions to enable it to retrieve relevant regional weather information for the user. Jelly Bean (API16) introduced the “Read_call_log”, “Write_call_log” and

Users are not in the habit of checking the permissions for which an app is authorised after a platform upgrade
“Read_external_storage” permissions, any of which might easily be misused to realise an attacker’s chosen objectives. Our malicious app could, for example, also request the “Read_call_log” and “Read_external_storage” permissions. Now if this app is installed by a user on a version of Android prior to Jelly Bean, say Ice Cream Sandwich (API15), then these last two permissions will not be defined by the platform (because they don’t yet exist) and so will not be recognised by the platform or authorised by the user during installation. The user will see only the requests for the “Internet” and “Access_coarse_location” and is likely to authorise what are usual permission requests for this kind of app.

When the user’s device receives an upgrade to Jelly Bean, upon restart, the app’s permission requests will be re-evaluated along with all other installed apps. At this time the two new permissions will be defined, and the platform will automatically authorise them without the user’s knowledge. The app now has access to functionality that was never authorised by the user and that the user initially is completely unaware of. It is trivial for an app to detect the platform version it is running on, and to modify its behaviour based on the result of that detection. In this way it can behave well on Ice Cream Sandwich and can then abuse the new permissions when running on Jelly Bean. Combined with the legitimately authorised “Internet” permission, the malicious app in this example could extract data from the call log and external storage and then transmit it across the internet to a machine controlled by the malware author.

The only way for the user to find out that new permissions have been assigned is to review the permissions listed in the app info screen. On Jelly Bean this would now show four permissions where previously there were two. Users, however, are not in the habit of checking the permissions for which an app is authorised after a platform upgrade. In fact, many users pay little attention in the first place. Dormant permissions mean that even those users that do pay attention during install will be ignorant of changes made when the platform upgrade awakens those dormant permissions.

Unfortunately the situation is complicated further by the fact there are times when the app info screen displays incorrect information and does not report the permissions an app is authorised to use. This complication comes from the manner in which the Android platform deals with developer-defined permissions and the order in which applications are installed.

More detailed information about this complication, the weakness described above and the Android permissions architecture in general can be found in my full dissertation.

About the authors

Having previously managed both computer networks and teams of technical individuals, most recently a software development team, James Sellwood currently works as a senior consultant at Consult Hyperion. Much of his work focuses on the application of information security principles, and a keen eye for detail, to contactless and mobile payment systems for various financial and telecommunications clients.

Jason Crampton has a BSc in Mathematics (University of Manchester) and a PhD in Computer Science (University of London). He has worked in the Information Security Group at Royal Holloway, University of London, since 2002, where he is a professor of information security. His research focuses on access control in multi-user computer systems.