Permissions

Contents

This document extends the higher-level Applications[1] and Security[2] design documents to go into more detail about the permissions framework.

Applications can perform many functions on a variety of user data. They may access interfaces that read data (such as contacts, network state, or the users location), write data, or perform actions that can cost the user money (like sending SMS). As an example, the Android operating system has a comprehensive

[1] https://em.pages.apertis.org/apertis-website/concepts/applications/
[2] https://em.pages.apertis.org/apertis-website/concepts/security/

manifest[3] that govern access to a wide array of functionality.

Some users may wish to have fine grained control over which applications have access to specific device capabilities, and even those that don't should likely be informed when an application has access to their data and services.

Terminology

Integrity, confidentiality and availability[4] are defined in the Security concept design.

Discussions of the security implications of a use-case in this document often mention the possibility of a *malicious* or *compromised* app-bundle. For brevity, this should be understood to cover all situations where malicious code might run with the privileges of a particular app-bundle, including:

- An app-bundle whose author or publisher deliberately included malicious code in the released version (a Trojan horse)
- An app-bundle whose author or publisher accidentally included malicious code in the released version, for example by using a maliciously altered compiler like XcodeGhost[5]
- An app-bundle where there is no directly malicious code in the released version, but there is a security vulnerability that an attacker can exploit to run malicious code of their choice with the privileges of the app-bundle

Scope of this document

This document aims to define a general approach to permissions, so that future work on a particular feature that requires permissions only requires the designer of that feature to define a permission or a series of permissions, and does not require the designer of the feature to design the entire permissions framework.

This document also aims to define permissions for basic features that are already present in Apertis and already well-understood. For example, access to external and shared storage is in-scope.

This document does not aim to define permissions for features that are not already present in Apertis, or that are not already well-understood. For example, defining detailed permissions for egress filtering[6] is out of scope.

The permissions framework is not intended to cover all the needs of built-in application bundles. See non use cases below. However, some classes of built-in application bundles are anticipated to be implemented by every Apertis vendor as a means of product differentiation, and these could benefit from having

[3] http://developer.android.com/reference/android/Manifest.permission.html
[4] https://em.pages.apertis.org/apertis-website/concepts/security/#integrity-confidentiality-and-availability
[5] https://en.wikipedia.org/wiki/XcodeGhost
[6] https://em.pages.apertis.org/apertis-website/concepts/egress_filtering/

a shortcut way to request particular permissions; these use cases are marked below.

Use cases

Internet access

A general-purpose Internet application like a web browser might require full, unfiltered Internet access, including HTTP, HTTPS, DNS, WebSockets and other protocols.

A podcast player might require the ability to download arbitrary files via HTTP using a service like the Apertis Newport download manager, but might not require any other Internet access.

A simple game might not require any network access at all, or might only require the indirect network access (launching URIs) that can be obtained by communicating with the Didcot content handover[7] service.

Many intermediate levels of Internet access are possible, but for the purposes of this document we do not consider them. See the Egress filtering design notes on the Apertis wiki[8] for initial work on finer-grained control.

Security implications

An application with Internet access might be compromised by malicious inputs from the Internet (integrity failure). If an application cannot contact the Internet, we can be confident that it cannot be subject to this, although it could still be compromised by malicious content that is downloaded locally and passed to it via content handover[9].

If an application with Internet access is compromised either remotely or by opening malicious local content, it could be induced to send private data to an attacker-controlled server (a confidentiality failure). This attack applies equally to applications with access to a download manager like Newport, because the private data could be encoded in the URI to be downloaded. If the download manager offers control over request headers such as cookies or the HTTP `Referer`, the private data could also be encoded in those. Applications that can request URIs via a content handover[10] service could also be susceptible to this attack, but only if the content handover service will pass them to a handler without user interaction, *and* the handler for those URIs will fetch the URI without user interaction.

If an application with Internet access is compromised, but does not already contain malicious code to carry out actions of the attacker's choice (the *payload*),

[7] https://em.pages.apertis.org/apertis-website/concepts/content_hand-over/
[8] https://em.pages.apertis.org/apertis-website/concepts/egress_filtering/
[9] https://em.pages.apertis.org/apertis-website/concepts/content_hand-over/
[10] https://em.pages.apertis.org/apertis-website/concepts/content_hand-over/

a common technique is to download a payload from a server controlled by the attacker. In particular, this allows an attacker to alter the payload over time according to their current requirements, for example to form a botnet that can be used for multiple purposes. An application with access to a download manager like Newport is equally susceptible to this, even if it cannot access the Internet itself, because it can ask Newport to download the new payload from the attacker's server. However, an application that can only use a content handover[11] service is not susceptible to this attack, because such applications are not allowed to see the result of the HTTP request.

In other systems

In Android, the web browser and podcast player described in the use-cases above would have the `INTERNET` permission, but the game would not. Using the Android DownloadManager service (equivalent to Newport) also requires `INTERNET` permission, because it enables most of the same attacks as direct Internet access.

In Flatpak, the web browser would have the `shared=network` permission and the game would not. The game could still send requests to the URI-opening portal: assuming that only one web browser is installed, the URI-opening portal would normally pass on HTTP and HTTPS URIs to a web browser without user consent (with the result that the browser makes `GET` requests to the appropriate web server), but prompt the user before passing other URIs to the URI handler. The podcast player could have `talk` access to a D-Bus service equivalent to Newport, but as noted above, that would be essentially equivalent to arbitrary HTTP access in any case.

In iOS, all app-bundles have Internet access.

Geolocation

A navigation app-bundle needs to know the precise location of the vehicle, but an app-bundle to suggest nearby restaurants might only need to know the location within a few miles, and an e-book reader does not need to know the location at all.

Security implications

The user's geographical location is sensitive information, especially if it is precise, and is valuable to criminals. In some cases disclosing it would be a threat to personal safety.

In other systems

[11] https://em.pages.apertis.org/apertis-website/concepts/content_hand-over/

In Android the navigation app-bundle would have `ACCESS_FINE_LOCATION`, the restaurant guide would have `ACCESS_COARSE_LOCATION` and the e-book reader would have neither.

In Flatpak, the user is asked for permission to use geolocation the first time it is used, with the option to remember that permission for all future requests. The app-bundle is not required to declare in advance whether it might use geolocation.

In iOS, two forms of geolocation can be requested, by using `NSLocationAlwaysUsageDescription` or `NSLocationWhenInUseUsageDescription`.

Initiating a phone call

A contact management application might wish to initiate a phone call without further user consent, for example when the user taps a phone icon next to a contact.

An application that is only tangentially related to phone calls, such as an app-bundle to suggest nearby restaurants, might not wish to request permission to do that. Instead, it could initiate a phone call by launching an appropriate `tel:` URI, which would normally result in a built-in application or a platform service popping up a call dialog with buttons to initiate the call or cancel the transaction, the same as would happen on selecting a `tel:` link in a web browser.

An e-book reader does not need to initiate phone calls at all.

Security implications

If an app-bundle can initiate calls without user consent, this will result in the user's microphone being connected to the call recipient, which is a confidentiality (privacy) failure. Making undesired calls can also cost the user money, and in particular a malicious app author might place calls to a premium rate number that pays them.

In other systems

In Android, the contact management app-bundle would have the `CALL_PHONE` permission. The restaurant guide and the e-book reader would not, but would still be able to launch an intent that results in the system phone dialler app being shown, giving the user the opportunity to confirm or cancel.

In Flatpak, the contact management app-bundle would have `talk` access to a D-Bus service offering immediate phone dialling, for example the Telepathy Account Manager, or to a group of services, for example Telepathy. The restaurant guide and the e-book reader would not, but would be able to launch a `tel:` URI, which would be handled in much the same way as the Android intent.

In iOS, user-installable app bundles would presumably launch `tel:` URIs. There does not appear to be a way for a non-platform-level component to dial phone numbers directly.

Shared file storage

A media player with a gallery-style user experience might require the ability to read media files stored on external storage (a USB thumb drive or externally-accessible SD card), or in a designated shared area[12].

Similarly, a media player might require access to media indexing and browsing as described in the Media Management concept design[13].

A podcast player might wish to store downloaded podcasts on external storage devices or in the shared storage area so that media players can access them.

Security implications

App-bundles with write access to this shared storage can modify or delete media files; if this is done inappropriately, that would be an availability or integrity failure. App-bundles with read access can observe the media that the user consumes, which could be considered privacy-sensitive; uncontrolled access would be a confidentiality failure. Malicious app-bundles with write access could also write malformed media files that were crafted to exploit security flaws in other app-bundles, in the platform, or in other devices that will read the same external storage device, leading to an integrity failure.

In other systems

In recent Android, the `READ_EXTERNAL_STORAGE` permission is required (the shared area on Android devices was traditionally a removable SD card, leading to the name of the relevant permission and APIs, even though in more recent devices it is typically on non-removable flash storage). In older Android, that permission did not exist or was not enforced.

Similarly, the `WRITE_EXTERNAL_STORAGE` permission governs writing; that permission was always enforced, but is very widely requested.

In Flatpak, any directory of interest can be mapped into the filesystem namespace of sandboxed processes, either read-only or read/write, via the `filesystems` metadata field. Values like `xdg-music` and `xdg-download/Podcasts` make common use cases relatively straightforward, and provide considerably finer-grained control than in Android.

In iOS, access to media libraries is mediated by the `NSAppleMusicUsageDescription` and `NSPhotoLibraryUsageDescription` metadata fields.

[12] https://em.pages.apertis.org/apertis-website/concepts/application-layout/#shared-data
[13] https://em.pages.apertis.org/apertis-website/concepts/media-management/

Launcher

This use-case is only applicable to built-in app-bundles.

A vendor-specific application launcher, such as the Mildenhall Launcher[14] in the Apertis Mildenhall reference user interface, needs to list all the application entry points on the system together with their metadata. It also needs to launch those entry points on-demand.

Security implications

Holding this permission negates the Apertis platform's usual concept of application list privacy[15]: an app-bundle with this permission can enumerate the entry points, which is valuable if an attacker wishes to identify particular user (fingerprinting). If unintended app-bundles gain this access, it is a confidentiality failure.

In other systems

Android does not appear to restrict the visibility of other app-bundles.

Flatpak app-bundles can only observe the existence of other app-bundles if their D-Bus filtering is configured to be able to `see` their well-known names.

iOS restricts the visibility of other app-bundles, although fingerprinting[16] can be carried out by abusing inter-app communication. Because iOS is a single-vendor system, the security mechanisms used by platform components and by the equivalent of our built-in app bundles do not have public documentation.

Settings

This use-case is probably only applicable to built-in app-bundles.

Suppose a vendor has a system preferences application[17] that provides an overview of all system settings[18], user settings[19] and app settings[20], such as the Mildenhall Settings[21] application bundle in the Apertis Mildenhall reference user interface. That application needs to list the app settings belonging to all

[14] https://gitlab.apertis.org/hmi/mildenhall-launcher
[15] https://em.pages.apertis.org/apertis-website/concepts/application-entry-points/#security-and-privacy-considerations
[16] https://arxiv.org/abs/1605.08664
[17] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#user-interface
[18] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#system-settings
[19] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#user-settings
[20] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#app-settings
[21] https://gitlab.apertis.org/hmi/mildenhall-settings

store and built-in app-bundles, and needs the ability to change them, without prompting the user.

Security implications

Holding this permission negates the Apertis platform's usual concept of application list privacy[22], similar to the Launcher use case.

Unconstrained settings changes are also very likely to allow arbitrary code execution with the privileges of other components that trust those settings, which would be a serious integrity failure if carried out by an attacker.

In other systems

In Android, the `CHANGE_CONFIGURATION` permission grants the ability to change system configuration in some limited ways, and the `WRITE_SETTINGS` permission grants the ability to carry out more settings changes.

In Flatpak when used with GNOME, granting write access to `dconf` (by making its files readable in the sandbox, and granting `talk` access to the dconf service) gives unconstrained access to all settings.

Because iOS is a single-vendor system, the security mechanisms used by platform components and by the equivalent of our built-in app bundles do not have public documentation.

Restricted subsets of settings

A photo viewer might have an option to set a particular photo as "wallpaper". A travel-related app-bundle might have an option to set the time zone, and media player might have options to change audio parameters. An e-book reader does not require the ability to do any of those.

Security implications

In general, these subsets of settings are chosen so that an attacker changing them would be an annoyance rather than a serious integrity failure, mitigating the attacks that are possible in the use-case above. However, the effect of changing a setting is not always immediately obvious: for example, setting untrusted images as wallpaper could lead to a more serious integrity failure if there is an exploitable flaw in an image decoder used by the platform component or built-in app-bundle that displays the wallpaper.

In other systems

[22] https://em.pages.apertis.org/apertis-website/concepts/application-entry-points/#security-and-privacy-considerations

In Android, the photo viewer might have the `SET_WALLPAPER` and `SET_WALLPAPER_HINTS` permissions, the travel-related app-bundle might have `SET_TIME_ZONE`, and the media player might have `MODIFY_AUDIO_SETTINGS`.

Flatpak does not currently have portals for these, but a Flatpak app-bundle could be given `talk` access to a D-Bus service that would allow these actions.

iOS does not appear to provide this functionality to third-party app-bundles.

Granting permission on first use

The author of a hotel booking app-bundle includes a feature to locate nearby hotels by using the Apertis geolocation API. Because users are more likely to grant permission to carry out privacy-sensitive actions if they can understand why it is needed[23], the app author does not want the Apertis system to prompt for access to the geolocation feature until the user actively uses that particular feature.

Not granting permission on first use

Conversely, an automotive vendor wishes to minimize driver distraction in order to maximize safety. When the same hotel booking app-bundle attempts to use geolocation while the vehicle is in motion, the platform vendor might want the Apertis system to **not** prompt for access to the geolocation feature, contrary to the wishes of the app author. Instead, the user should be given the opportunity to enable geolocation at a time when it is safe to do so, either during app-bundle installation or as a configuration/maintenance operation while the vehicle is stationary at a later time.

Note that those two use cases have contradictory expectations: this is a user experience trade-off for which there is no single correct answer.

In other systems

iOS prompts for permission to carry out each privileged operation at the time of first use.

Flatpak mostly does the same, but with some pragmatic exceptions: lower-level permissions, such as access to direct rendering devices for 3D games or direct access to the host filesystem, are implemented in a way that precludes that model. These are set up at installation time, and can be overridden by user configuration. When a Flatpak app is launched, it is given the level of access that was appropriate at launch time.

Android 6.0 and later has the same behaviour as iOS. Older Android versions configured all permissions at installation time, with a simple UX: the user must either accept all required permissions, or abort installation of the app. Some permissions, notably access to shared storage (the real or emulated SD card),

[23] https://savvyapps.com/blog/how-to-create-better-user-permission-requests-in-ios-apps

were implemented in a way that precluded runtime changes: app processes with access to shared storage ran with one or more additional Unix group IDs, granting them DAC permission to the appropriate areas of the filesystem.

Tightening control

Suppose that Apertis version 1 allows all app-bundles to query the vehicle model, but the Apertis developers later decide this is a privacy risk, and so Apertis version 2 restricts it with a permission. The app framework should be able to detect that an app-bundle was compiled for version 1, and behave as though that app-bundle had requested the necessary permission to query the vehicle model. It should not do that for an app-bundle compiled for version 2.

Security implications

App-bundles that were compiled for version 1 would still be able to carry out any attacks that were applicable before version 2 was released. This use-case is only applicable if those attacks are considered to be less serious than breaking backwards compatibility with older app-bundles.

In other systems

In Android, a simple integer "API level" is used to indicate the version of the Android API. Each app-bundle has a *minimum API level* and a *target API level*. The app framework enables various compatibility behaviours to make APIs resemble those that were present at the target API level; one of these compatibility behaviours is to behave as though app-bundles whose target API level is below a threshold had requested extra permissions. For example, Android behaves as though app-bundles with a target API level below 4 had requested `android.READ_PHONE_STATE`.

In Flatpak, app-bundles can specify a minimum Flatpak version. There is is currently no mechanism to specify a target API level, although one could be inferred from the runtime branch that the app-bundle has chosen to use, such as `org.freedesktop.Platform/1.4` or `org.gnome.Platform/3.22`.

In iOS, keys like [NSAppleMusicUsageDescription] are documented as behaving like permissions, but only if the app was linked on or after iOS 10.0.

Loosening control

Suppose Apertis version 1 restricts querying the vehicle paint colour with a permission, but the Apertis developers later decide that this does not need to be restricted, and Apertis version 2 allows all app-bundles to do that. The app framework should never prompt the user for that permission. If an app-bundle designed for version 1 checks whether it has that permission, the app framework should tell it that it does.

Security implications

This use-case is only applicable if the Apertis developers have decided that the security implications of the permission in question (in this example, querying the paint colour) are not significant.

In other systems

We are not aware of any permissions that have been relaxed like this in Android, Flatpak or iOS, but it would be straightforward for any of these frameworks to do so: they would merely have to stop presenting a user interface for that permission, and make requests for it always succeed.

Changing access

An Apertis user uses a Facebook app-bundle. The user wants their location at various times to appear on their Facebook feed, so they give the app-bundle permission to monitor his location, as in geolocation above.

Later, that user becomes more concerned about their privacy. They want to continue to use the Facebook app-bundle, but prevent it from accessing their new locations. They use a user interface provided by the system vendor, perhaps a system preferences application[24], to reconfigure the permissions granted to the Facebook app-bundle so that it cannot access their location.

Later still, that user wants to publish their location to their Facebook feed while on a road trip. They reconfigure the permissions granted to the Facebook app-bundle again, so that it can access their location again.

Security implications

This use-case is applicable if the user's perception of the most appropriate trade-off between privacy and functionality changes over time.

In other systems

Android 6.0 and later versions have a user interface[25] to revoke and reinstate broad categories of permissions. Older Android versions had a hidden control panel named App ops[26] controlling the same things at a finer-grained level (individual permissions), but it was not officially supported.

iOS allows permissions to be revoked or reinstated at any time via the Privacy page in its Settings app[27], which is the equivalent of the Apertis system

[24] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#user-interface

[25] https://www.howtogeek.com/230683/how-to-manage-app-permissions-on-android-6.0/

[26] https://www.theguardian.com/technology/2015/jun/09/google-privacy-apple-android-lockheimer-security-app-ops

[27] https://www.howtogeek.com/177711/ios-has-app-permissions-too-and-theyre-arguably-better-than-androids/

preferences application[28].

Potential future use-cases

Use cases described in this section are not intended to generate requirements in the near future, and are not described in detail here. We recommend that these use cases are expanded into something more detailed as part of design work on the relevant feature: for example, Bluetooth permissions should be considered as part of a more general Bluetooth feature design task.

However, as input to the design of the general feature of permissions, it might be instructive to consider whether a proposed implementation could satisfy the requirements that these use-cases are conjectured to have.

Because these use-cases have not been examined in detail, it is possible that future work on them will result in the conclusion that they should be outside the scope of the permissions framework described in this document.

Audio playback

A music player requires the ability to play back audio while in the background. A video player might require the ability to play ongoing audio, but only while its window is in the foreground. An e-book reader might only require the ability to play short notification sounds while in the foreground, or might not require any ability to play sounds at all. A voice-over-IP calling client requires the ability to play audio with an elevated priority while a call is in progress, pre-empting other audio players.

We recommend that these and related use cases are captured in detail as part of the design of the Apertis audio manager.

Security implications

Uncontrolled audio playback seems likely to cause driver distraction. Additionally, if all applications can play back audio with a priority of their choice, a malicious app-bundle could output silence at a high priority as a denial of service attack (a failure of availability).

In other systems

In Android and iOS, audio playback does not require special permissions.

In Flatpak, audio playback currently requires making the PulseAudio socket available to the sandboxed app, which also enables audio recording and control. Finer-grained control over audio is planned for the future.

[28] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#user-interface

Audio recording

A memo recorder requires the ability to record audio. A voice-over-IP calling client also requires the ability to record audio. Most applications, including most of those that play back audio, do not.

We recommend that these and related use cases are captured in detail as part of the design of the Apertis audio manager.

Security implications

An app-bundle that can record audio could record private conversations in the vehicle (a failure of confidentiality).

In other systems

In Android, audio recording requires the `RECORD_AUDIO` permission.

In Flatpak, audio recording currently requires making the PulseAudio socket available to the sandboxed app, which also enables audio playback and control.

In iOS, audio recording is mediated by `NSMicrophoneUsageDescription`.

Bluetooth configuration

A system preferences application[29], or a separate Bluetooth control panel built-in app-bundle, might require the ability to reconfigure Bluetooth in detail and communicate with arbitrary devices.

A less privileged app-bundle, for example one provided by the manufacturer of peripheral devices like FitBit, might require the ability to pair and communicate with those specific Bluetooth devices.

A podcast player has no need to communicate with Bluetooth devices at all.

Security implications

For the control panel use-case, communicating with arbitrary devices might be an integrity failure if the app-bundle can reconfigure the device or edit data stored on it, or a confidentiality failure if the app-bundle can read sensitive data such as a phone's address book. The ability for untrusted app-bundles to view MAC addresses and other unique identifiers would also be a privacy problem.

The device-specific use case is a weaker form of the above, mitigating the confidentiality and integrity impact.

[29] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#user-interface

In other systems

In Android, the `BLUETOOTH` permission allows an app-bundle to communicate with any Bluetooth device that is already paired. This is stronger than is needed for a device-specific app-bundle. The `BLUETOOTH_ADMIN` permission additionally allows the app-bundle to pair new Bluetooth devices.

In Flatpak, full access could be achieved by configuring Flatpak's D-Bus filter to allow `talk` access to BlueZ. There is currently no implementation of partial access; this would likely require a Bluetooth portal service.

In iOS, the NSBluetoothPeripheralUsageDescription[30] metadata field controls access to Bluetooth, which appears to be all-or-nothing. User consent is requested the first time this permission is used, with the metadata field's content included in the prompt.

Calendar

A general-purpose calendar/agenda user interface similar to GNOME Calendar[31] or the AOSP Calendar[32] requires full read/write access to the user's calendar.

A calendar synchronization implementation, for example to synchronize with calendar events stored in Google Calendar, Windows Live or OwnCloud, requires full read/write access to its subset of the user's calendar. For example, a Google Calendar synchronization app-bundle should have access to Google calendars, but not to Windows Live calendars.

A non-calendaring application like an airline booking app-bundle might wish to insert events into the calendar without further user interaction, or it might wish to insert events into the calendar in a way that presents them for user approval, for example by submitting a vCalendar file for content handover[33].

A podcast player has no need to interact with the calendar at all.

Security implications

The general-purpose user interface described above would have the ability to send calendar events to a third party (a confidentiality failure) or to edit or delete them (an integrity failure).

The calendar synchronization example is a weaker form of the user interface use-case: if malicious, it could cause the same confidentiality or integrity failures, but only for a subset of the user's data.

[30] https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW20

[31] https://wiki.gnome.org/Apps/Calendar

[32] https://fossdroid.com/a/standalone-calendar.html

[33] https://em.pages.apertis.org/apertis-website/concepts/content_hand-over/

If the airline booking app-bundle described above has the ability to insert calendar events without user interaction, a malicious app-bundle could insert misleading events, an integrity failure; however, it would not necessarily be able to break confidentiality.

If the airline booking app-bundle operates via content handover, intents, portals or a similar mechanism that will result in user interaction, a malicious app-bundle cannot insert misleading events without user action, avoiding that integrity failure (at the cost of a more prescriptive UX).

In other systems

In Android, the `READ_CALENDAR` and `WRITE_CALENDAR` permissions[34] are suitable for the general-purpose calendar use case. Sync adapters[35] receive different access; it is not clear from the Android documentation whether their restriction to a specific subset of the calendar is enforced, or whether sync adapters are trusted and assumed to not attack one another. Applications that do not have these permissions, such as the hotel booking use-case above, can use calendar intents[36] to send or receive calendar events, with access mediated through a general-purpose calendar user interface that is trusted to behave according to the user's intention. There is no way to prevent an app-bundle from using those intents at all.

In Flatpak, a general-purpose calendar might be given `talk` access to the `evolution-data-server` service. There is currently no calendar portal, but when one is added it will presumably be analogous to Android intents.

In iOS, the [NSCalendarsUsageDescription] metadata field controls access to calendars. User consent is requested the first time this permission is used, with the metadata field's content included in the prompt.

Contacts

The use cases and security implications for contacts are analogous to those for the calendar and are not discussed in detail here.

In other systems

Android contact management[37] is analogous to calendaring, using the `READ_CONTACTS` and `WRITE_CONTACTS` permissions or contact-specific intents.

In Flatpak, as with contacts, a general-purpose contacts app-bundle might be given `talk` access to the `evolution-data-server` service. There is currently no

[34] https://developer.android.com/guide/topics/providers/calendar-provider.html#manifest
[35] https://developer.android.com/guide/topics/providers/calendar-provider.html#sync-adapter
[36] https://developer.android.com/guide/topics/providers/calendar-provider.html#intents
[37] https://developer.android.com/guide/topics/providers/contacts-provider.html

contacts portal, but when one is added it will presumably be analogous to Android intents.

[iOS contact management][NSContactsUsageDescription] is analogous to iOS calendaring.

Inter-app communication interfaces

Inter-app communication has not been designed in detail, but the draft design on the Apertis wiki suggests that it might be modelled in terms of interface discovery[38], with app-bundles able to implement "public interfaces" that are made visible to other app-bundles. The draft design has some discussion of how restricting interface providers[39] might be carried out by app-store curators.

Additionally, if app-bundles export public interfaces, this might influence whether other applications are allowed to communicate with them: if a particular public interface implies that other app-bundles will communicate directly with the implementor, then the implementor's AppArmor profile and other security policies must allow that. A sharing[40] feature similar to the one in Android is one possible use-case for this.

We recommend that this topic is considered as one or more separate concept designs, with its security implications considered at the same time. This is likely to be more successful if a small number of specific use-cases are considered, rather than attempting to define a completely abstract and general framework.

In Android, any app-bundle can define its own intents. If it does, those intents can be invoked by any other app-bundle that holds appropriate permissions, and it is up to the implementor to ensure that that is a safe thing to do.

In Flatpak, app-bundles that will communicate via D-Bus can be given `talk` access to each other. If this is done, it is up to the app-bundles to ensure that they do not carry out unintended actions in response to D-Bus method calls.

In iOS, any app-bundle can define non-standard URI schemes that it will handle, and these non-standard URI schemes are the basis for inter-app communication. There is no particular correlation between the URI scheme and the app's identity (the iOS equivalent of our bundle IDs), and there have been successful attacks against this, including the URL masque attack[41] identified by FireEye.

Continuing to run in the background

Agents[42] do not show any graphical windows, so to be useful they must always run in the background.

[38] https://em.pages.apertis.org/apertis-website/concepts/interface_discovery/
[39] https://em.pages.apertis.org/apertis-website/concepts/interface_discovery/#Restricting_who_can_advertise_a_given_interface_2
[40] https://em.pages.apertis.org/apertis-website/concepts/sharing/
[41] https://www.fireeye.com/blog/threat-research/2015/04/url_masques_on_apps.html
[42] https://em.pages.apertis.org/apertis-website/concepts/applications/#start

Graphical programs that have windows open, but no windows visible to the user, might be terminated by the application framework. The author of a graphical program that needs to be available without delay might wish to request that it is not terminated.

Security implications

Background programs consume resources, impacting availability (denial of service). A background program that has other permissions might make use of them without the user's knowledge: for example, if a restaurant guide can track the user's location, this can be mitigated by only allowing it to run, or only allowing it to make use of its permissions, while it is (or was recently) visible, so that the user can only be tracked by the guide's author at times when they are aware that this is a possibility.

Users might wish to be aware of which graphical programs have this property, and user interfaces for managing permissions might display it in the same context as other permissions, but it is not a permission in the sense that it is used to generate security policies. Accordingly, it should potentially be handled outside the scope of this document.

Future work on this topic is tracked in Apertis task T3438[43] and its future subtasks.

In other systems

Android does not have permissions that influence its behaviour for background programs.

Flatpak does not currently attempt to monitor background programs or force them to exit.

iOS manages background programs via the [UIBackgroundModes] and UIApplicationExitsOnSuspend[44] metadata fields. NSSupportsAutomaticTermination[45] is analogous, but is for desktop macOS.

Running on device startup

An agent[46] might be run on device startup.

A typical graphical program has no need to start running on device startup.

[43] https://phabricator.apertis.org/T3438

[44] https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/iPhoneOSKeys.html#//apple_ref/doc/uid/TP40009252-SW23

[45] https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW13

[46] https://em.pages.apertis.org/apertis-website/concepts/applications/#start

A graphical program that is expected to be frequently but intermittently used might be pre-loaded (but left hidden) on device startup.

The security implications are essentially the same as continuing to run in the background.

Users might wish to be aware of which graphical programs have this property, and user interfaces for managing permissions might display it in the same context as other permissions, but it is not a permission in the sense that it is used to generate security policies. Accordingly, it is treated as outside the scope of this document.

We suggest that this should be handled alongside continuing to run in the background.

In other systems

In Android, a graphical program or service that runs in the background would have the `RECEIVE_BOOT_COMPLETED` permission, which is specifically described as covering performance and not security.

Flatpak does not natively provide this functionality.

iOS manages autostarted background programs via certain values of the [UIBackgroundModes] metadata field.

Non-use-cases

The following use cases are specifically excluded from the scope of this document.

App's own data

Each app-bundle should be allowed to read and write its own data, including its own app settings[47]. However, this should not need any special permissions, because it should be granted to every app-bundle automatically: accordingly, it is outside the scope of this document. App settings are part of the scope of the Preferences and Persistence[48] concept design, and other per-app private data are in the scope of the Applications[49] concept design.

Similarly, programs from each app-bundle should be allowed to communicate with other programs from the same app-bundle (using any suitable mechanism, including D-Bus) without any special permissions, with the typical use-case being a user interface communicating with an associated agent[50]. Because it does not require special permissions, that is outside the scope of this document.

[47] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/#app-settings
[48] https://em.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/
[49] https://em.pages.apertis.org/apertis-website/concepts/applications/
[50] https://em.pages.apertis.org/apertis-website/concepts/applications/#start

Platform services

This permissions framework is not intended for use by platform services, regardless of whether they are upstream projects (such as systemd, dbus-daemon and Tracker), developed specifically for Apertis (such as the Newport download manager), or developed for a particular vendor. Platform services should continue to contain their own AppArmor profiles, polkit rules and other security metadata.

Built-in app-bundles with specialized requirements

This permissions framework is not intended for use by built-in application bundles with specialized or highly-privileged requirements, such as a built-in application that communicates directly with specialized hardware. These built-in application bundles should have their own AppArmor profiles, polkit rules and other security metadata.

Driving cameras

Some vehicles have external cameras for purposes such as facilitating reversing, watching for hazards in the vehicle's blind spots, or improving night vision by using thermal imaging.

Our understanding is that images from these cameras should only be made available to platform components or to specialized built-in app-bundles, so they are outside the scope of this document.

Infotainment cameras

Android and iOS mobile phones and tablets typically have one or more cameras directed at the user or their surroundings, intended for photography, videoconferencing, augmented reality and entertainment. Our understanding is that this is not a normal use-case for an automotive operating system that should minimize driver distraction.

If a vehicle does have such cameras, their use cases and security implications are very similar to audio recording, so we believe there is no need to describe them in detail in this document.

App-specific permissions

In Android, any app-bundle can declare its own unique permissions namespaced by its author's reversed domain name, and any other app-bundle can request those permissions. It is not clear how an app-store vendor can be expected to make an informed decision about whether those requests are legitimate.

If an app-bundle signed by the same author requests one of these permissions, it is automatically granted; Android documentation recommends this route.

If an app-bundle by a different author that requests one of these app-specific permissions is installed, a description provided by the app-bundle that declared the permission is shown to the user when they are choosing whether to allow the requesting app-bundle to be installed. If the requesting app-bundle is installed before the declaring app-bundle, then its request to use that permission is silently denied.

Flatpak does not directly have this functionality, although cooperating app-bundles can be given `talk` access to each other's D-Bus well-known names.

iOS does not appear to have this functionality.

We recommend that this feature is not considered in the short term.

General notes on other systems

Specific permissions corresponding to those for which we see a need in Apertis are covered in the individual use cases above. This section describes other operating systems and app frameworks in more general terms.

Android

Android includes permissions in its XML manifest file.

- Introduction[51]
- Permission API reference[52]
- Permission group API reference[53]
- Declaring that a permission is needed[54]

Android apps can declare new permissions in the XML manifest.

- Permission element[55]
- Permission group element[56]
- Permission tree element[57]

Since Android 6.0, it is possible to request additional permissions (not declared in the manifest) at runtime.

Permissions not described in this document

The following access permissions, available as of API level 25, do not match any use-case described in this document. Deprecated and unsupported permissions have been ignored when compiling this document.

[51] https://developer.android.com/guide/topics/manifest/manifest-intro.html
[52] https://developer.android.com/reference/android/Manifest.permission.html
[53] https://developer.android.com/reference/android/Manifest.permission.html
[54] https://developer.android.com/guide/topics/manifest/uses-permission-element.html
[55] https://developer.android.com/guide/topics/manifest/permission-element.html
[56] https://developer.android.com/guide/topics/manifest/permission-group-element.html
[57] https://developer.android.com/guide/topics/manifest/permission-tree-element.html

Normal permissions:

- ACCESS_LOCATION_EXTRA_COMMANDS
- ACCESS_NETWORK_STATE
- ACCESS_NOTIFICATION_POLICY
- ACCESS_WIFI_STATE
- ADD_VOICEMAIL
- BATTERY_STATS
- BODY_SENSORS
- BROADCAST_STICKY
- CAMERA
- CHANGE_NETWORK_STATE
- CHANGE_WIFI_MULTICAST_STATE
- CHANGE_WIFI_STATE
- DISABLE_KEYGUARD
- EXPAND_STATUS_BAR
- GET_ACCOUNTS
- GET_ACCOUNTS_PRIVILEGED
- GET_PACKAGE_SIZE
- INSTALL_SHORTCUT
- KILL_BACKGROUND_PROCESSES
- NFC
- PROCESS_OUTGOING_CALLS
- READ_CALL_LOG
- READ_EXTERNAL_STORAGE
- READ_PHONE_STATE
- READ_SMS
- READ_SYNC_SETTINGS
- READ_SYNC_STATS
- RECEIVE_MMS
- RECEIVE_SMS
- RECEIVE_WAP_PUSH
- REORDER_TASKS
- REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
- REQUEST_INSTALL_PACKAGES
- SEND_SMS
- SET_ALARM
- SET_TIME_ZONE
- SET_WALLPAPER
- SET_WALLPAPER_HINTS
- TRANSMIT_IR
- USE_FINGERPRINT
- USE_SIP
- VIBRATE
- WAKE_LOCK
- WRITE_CALL_LOG

- WRITE_EXTERNAL_STORAGE
- WRITE_SETTINGS
- WRITE_SYNC_SETTINGS

Permissions described as not for use by third-party applications:

- ACCOUNT_MANAGER
- Several permissions starting with BIND_ that represent the ability to bind to the identity of a platform service, analogous to the ability to own platform services' D-Bus names in Apertis
- BLUETOOTH_PRIVILEGED
- Several permissions starting with BROADCAST_ that represent the ability to broadcast messages, analogous to the ability to own a platform service's D-Bus name and send signals in Apertis
- CALL_PRIVILEGED
- CAPTURE_AUDIO_OUTPUT
- CAPTURE_SECURE_VIDEO_OUTPUT
- CAPTURE_VIDEO_OUTPUT
- CHANGE_COMPONENT_ENABLED_STATE
- CLEAR_APP_CACHE
- CONTROL_LOCATION_UPDATES
- DELETE_CACHE_FILES
- DELETE_PACKAGES
- DIAGNOSTIC
- DUMP
- FACTORY_TEST
- GLOBAL_SEARCH, held by the global search framework to give it permission to contact every global search provider
- INSTALL_LOCATION_PROVIDER
- INSTALL_PACKAGES
- LOCATION_HARDWARE
- MANAGE_DOCUMENTS
- MASTER_CLEAR
- MEDIA_CONTENT_CONTROL
- MODIFY_PHONE_STATE
- MOUNT_FORMAT_FILESYSTEMS
- MOUNT_UNMOUNT_FILESYSTEMS
- PACKAGE_USAGE_STATS
- READ_FRAME_BUFFER
- READ_LOGS
- READ_VOICEMAIL
- REBOOT
- SEND_RESPOND_VIA_MESSAGE
- SET_ALWAYS_FINISH
- SET_ANIMATION_SCALE
- SET_DEBUG_APP
- SET_PROCESS_LIMIT

- `SET_TIME`
- `SIGNAL_PERSISTENT_PROCESSES`
- `STATUS_BAR`
- `SYSTEM_ALERT_WINDOW`
- `UPDATE_DEVICE_STATS`
- `WRITE_APN_SETTINGS`
- `WRITE_GSERVICES`
- `WRITE_SECURE_SETTINGS`
- `WRITE_VOICEMAIL`

Intents

Holding a permission is not required to use an *intent* that implicitly asks the user for permission, such as taking a photo by sending a request to the system camera application, which will pop up a viewfinder provided by the system camera application, allowing the user to either take a photo when they are ready, or cancel by pressing the Back button; if the user takes a photo, it is sent back to the requesting application as the result of the intent. This is conceptually similar to Flatpak portals.

Flatpak

Flatpak does not have a single flat list of permissions. Instead, its permissions are categorized according to the resource being controlled. Available permissions include:

- Hardware-accelerated graphics rendering via Direct Rendering Manager[58] devices
- Hardware-accelerated virtualization via Kernel-based Virtual Machine[59] devices
- Full access to the host's device nodes
- Sharing specific filesystem areas on a read-only or read/write basis
- Sharing the host's X11 socket (not used in production on Apertis)
- Sharing the host's Wayland socket (always available to graphical programs on Apertis)
- Full access to the host's D-Bus session bus
- Full access to the host's D-Bus system bus
- Full access to the host's PulseAudio socket
- Sharing the host system's network namespace (Internet and LAN access)
- Sharing the host system's IPC namespace (this does not control D-Bus or `AF_UNIX` sockets, but would allow the app-bundle to be treated as unconfined for the purposes of services that use Unix System V IPC[60] or POSIX message queues[61]

[58] https://en.wikipedia.org/wiki/Direct_Rendering_Manager
[59] https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine
[60] https://manpages.debian.org/svipc(7)
[61] https://manpages.debian.org/mq_overview(7)

- Sending and receiving messages to communicate with a specific D-Bus well-known name (`talk` access)
- Permission to own (provide) specific D-Bus well-known names (`own` access)

Portals

Flatpak portals[62] are similar to Android intents. These components expose a subset of desktop functionality as D-Bus services that can be used by contained applications: they are part of the security boundary between a contained app and the rest of the desktop session. The aim is for portals to get the user's permission to carry out actions, while keeping it as implicit as possible, avoiding an "are you sure?" step where feasible. For example, if an application asks to open a file, the user's permission is implicitly given by them selecting the file in the file-chooser dialog and pressing OK: if they do not want this application to open a file at all, they can deny permission by cancelling. Similarly, if an application asks to stream webcam data, the expected UX is for GNOME's Cheese app or a similar non-GNOME app to appear, open the webcam to provide a preview window so they can see what they are about to send, but not actually start sending the stream to the requesting app until the user has pressed a "Start" button. When defining the API "contracts" to be provided by applications in that situation, portal designers need to be clear about whether the provider is expected to obtain confirmation like this: in most cases we anticipate that it will be expected to do this.

If this sort of implicit permission is not feasible for a particular portal, it is possible for the portal implementation to fall back to a model similar to iOS, by asking the user for explicit consent to access particular data. Flatpak provides a portal-facing API (the *permissions store*) with which a portal can check whether the user already gave permission for particular operations, or store the fact that the user has now given permission. Each portal can define its own permissions, but app-bundles cannot normally do so.

There is currently no user interface for the user to review previously-granted permissions and revoke them if desired, but one could be added in future, again similar to iOS.

Unlike Android intents, different Flatpak portals use different mechanisms to send the result of a request to the portal back to the requesting app-bundle. For example, many portals send and receive small requests and results over D-Bus, but the file chooser makes the selected file available in a FUSE filesystem that is visible inside the Flatpak sandbox. This avoids having to stream the whole file over D-Bus, which could be very slow and inefficient, particularly the file is very large and the app will carry out random access within it (such as seeking within a video).

More information on Flatpak portals can be found in the article The flatpak

[62] https://github.com/flatpak/flatpak/wiki/Portals

security model, part 3[63].

iOS

The iOS 10 model for permissions is a hybrid of the intents/portals approaches, and the approach of pre-declaring Android permissions. Apps that need access to sensitive APIs (analogous to portals) must provide a description of why that access is required. This gives the app-store curator an opportunity to check that these permissions make sense, as with Android permissions. However, unlike Android, user consent is requested at the time the app tries to exercise that access, not during installation. The given description is included in the prompt, and can be used to justify why access is needed.

There is also a user interface for the user to review previously-granted permissions, and revoke them if desired.

Permissions not described in this document

The usage descriptions that are the closest equivalent of permissions in iOS appear to be a subset of the Cocoa `Info.plist` keys[64], where `Info.plist` is the iOS equivalent of our application bundle metadata[65]. They exist in the same namespace as non-permission-related keys such as human-readable copyright notices.

Usage descriptions not corresponding to a use-case in this document include:

- `NSCameraUsageDescription`
- `NSHealthShareUsageDescription`
- `NSHealthUpdateUsageDescription`
- `NSHomeKitUsageDescription`
- `NSMotionUsageDescription` (accelerometer)
- `NSRemindersUsageDescription`
- `NSSiriUsageDescription`
- `NSSpeechRecognitionUsageDescription`
- `NSVideoSubscriberAccountUsageDescription`

[63] https://blogs.gnome.org/alexl/2017/01/24/the-flatpak-security-model-part-3-the-long-game/

[64] https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html

[65] https://em.pages.apertis.org/apertis-website/concepts/application-bundle-metadata/