Multiuser

Contents

This document describes how multiple users are expected to use the Apertis system, and works mostly as a guide and recommendations to help designing the system. It is intended to act as an "umbrella"document covering the multi-user topic in general, and will be supplemented by more concrete documents describing particular use-cases and recommendations for how those use-cases can be addressed.

The driving force behind having a multi-user system is to allow customization of the system. A system may have multiple users who would be frustrated by customizations done by each other to the system's look and feel and even to data such as playlists. Having multiple users allows each to customize their own interface.

Depending on OEM and consumer requirements, multi-user systems can potentially also provide personal files and online accounts for each user.

Terminology and concepts

"user"vs. "uid"

In a Unix system, users are typically identified by a numeric user ID, often abbreviated "uid". A uid can represent a person, a system facility, multiple people, or even an application (as in Android).

Because these do not correspond 1:1 in some designs, it is important to be clear which one is under discussion. In this document, the jargon term *uid* or *user ID* is used to refer to a Unix user identifier, while *user* or *person* is used to refer to a human using the system.

User account refers to any abstract representation of the user within the system. This is most commonly a uid, matching the original Unix design. However, systems can exist with multiple uids per user account, such as Android, in which each (user account, app) pair has a uid. Conversely, systems can exist with multiple user accounts sharing a uid, such as SteamOS (in which one uid runs the Steam Big Picture UI, and users log in to it with separate Steam accounts).

The canonical form of a Unix uid is numeric, but for ease of reference, a short lower-case textual *username* may be used to refer to a uid. For example, it is common to talk about system users named "root"and "backup", but the real identities of these users within the system are the corresponding numeric uids 0 and 34; the usernames are merely for convenience and mnemonic value.

Trusted components

A *trusted* component is a component that is technically able to violate the security model (i.e. it is relied on to enforce a privilege boundary), such that errors or malicious actions in that component could undermine the security model. The *trusted computing base* is the set of trusted components. This is independent of its quality of implementation –it is a property of whether the component is relied on in practice, and not a property of whether the component is *trustworthy*, i.e. safe to rely on. For a system to be secure, it is necessary that all of its trusted components must be trustworthy.

One subtlety of Apertis'app-centric design is that there is a trust boundary between applications even within the context of one user. As a result, a multi-user design has two main layers in its security model: system-level security that protects users from each other, and user-level security that protects a user's apps from each other. Where we need to distinguish between those layers, we will refer to the *TCB for security between users* or the *TCB for security between apps* respectively.

System services

A *system service* is a service that, conceptually, runs on behalf of the whole computer, without a division between users. In designs where each user has a distinct uid, system services run under a system uid, either root (the most privileged uid) or a special unprivileged uid per service or group of services; they do not run with the uid of any particular user.

This term does not necessarily imply anything about whether the service is considered to be "part of the operating system", or whether it is part of a pre-installed or user-installable application bundle as discussed in the Applications Design document. However, because system services can accept requests from multiple users, any system service that will handle users'private data must be trusted to impose a privilege boundary.

Examples of system services commonly present in Linux systems include ConnMan, NetworkManager, BlueZ, udisks and the D-Bus system bus.

User services

A *user service* is a service that runs on behalf of a particular user. In designs where each user has a distinct uid, each user's user services typically run under that same uid; in designs like SteamOS where all users share a single generic uid representing "all users", user services would typically share that same uid.

Examples of user services commonly present in Linux systems include dconf, gvfs, Tracker, Tumbler and the D-Bus session bus.

Identifying a process as a user service is independent of whether it is treated as part of the Apertis platform and independent of any particular application (such as the user services mentioned above), or treated as part of an application bundle ("agents"associated with apps).

Multi-seat (logind seats)

In the context of a multi-user system, a *seat* is a collection of display and input devices, optionally linked to other devices such as a USB socket or optical drive, intended to be used by one user at a time. Typical PCs only offer one seat, but a second graphics adapter, often connected via USB, can be used to add additional seats (a *multi-seat* system).

This jargon term is commonly used in Linux system services such as systemd-logind, the older ConsoleKit, and GDM. As an example, in the context of a car, it should be noted that it does not necessarily correspond precisely to the car's seats: for instance, in the common layout that places a single "head unit" touchscreen between the driver and front passenger, that touchscreen and any USB sockets adjacent to it would be treated as a single seat. If, for example, additional touchscreens were added behind the front seats for use by rear passengers, that would be a multi-seat system with 3 seats (front, rear left, rear right).

Apertis uses systemd-logind as a core system service, so where disambiguation is needed, we will refer to this as a *logind seat*.

Fast user switching

Many operating systems have the concept of *fast user switching*, which is described in Fast user switching: switching user without logging out. Following common usage, this document reserves the term "fast user switching"to refer to that particular multi-user model, even if some other model might be equally fast or faster in practice.

Requirements

Apertis is currently designed as a single-user system. There is one GUI session with full access to all preferences, apps and data, and a set of apps and user services with varying levels of sandboxing and privilege separation from each other within that session, running on top of system services whose privileges vary. The high-level requirement for this document is that this should be expanded to support multiple GUI users, each with their own private data and user services, running on top of similar system services.

This section contains a list of general requirements applicable to many multi-user systems.

- Multiple users should be able to use the system. Depending on the specific set of requirements, this could involve concurrent use, or one user at a time.
- When the user logs in to a newly started system, they should find the same applications they had left open last time they shut down the system, and in the same state. See Returning to previous state for discussion of this topic.
- Some data is private to each user. Depending on the specific set of requirements, this could include:
    - Settings
    - Address book
    - Browser history
    - Application icons
    - Arrangement of icons in the app launcher
    - Account data for web services
    - Playlists
- Some data is shared between users. Depending on the specific set of requirements, this could include:
    - Applications (from the store)
    - Media library (music, videos)
- Depending on the specific set of requirements, switching users at runtime could be supported. Where it exists, this shall be performed with a smooth transition, with no visual flickering. User switching should not take more than 5 seconds. See [Switching between users] for discussion of this topic.
- A subset of features are considered to be core functionality, and must not be disturbed by switching between users: they must remain available before, during and after any transition between users. The set of core functionality could vary by device; in this document we mainly use music playing and navigation as examples of this category. See Preserving “core” functionality across user-switching for further discussion of this topic.
- The subset of features that are not disturbed while switching between users must not be limited to functionality that is considered to be “part of the operating system”. For example, it should be possible to place a user-installable player for a third-party music streaming service such as Spotify or last.fm in this category. Again, see Preserving “core”functionality across user-switching

- Depending on the specific set of requirements, peripheral hardware devices such as USB storage devices and paired Bluetooth devices could either be shared across the entire system, or specific to a user. If they are shared, then they must be accessible to all users, with all users able to unmount/eject them.

- The authentication and user-switching user interface should not distract the driver more than is necessary; for instance, they should not ask security or authentication questions unless a decision is strictly required.

- The user privileges of the system should be visually obvious: if users have selected different personalizations such as colour schemes or themes, then the display should use a particular user's theme whenever it is acting on behalf of that user, and at no other time. This limits the risk that users will encounter undesired privacy consequences resulting from misunderstanding the system's privacy model.

Distinguishing between privacy levels in user-specific data

There are several possible categories of user-specific data.

Some user-specific data is private. For instance this might include email, browsing history, social media feeds. (Alice should not be able to read Bob's email, history, social media feeds and so on unless Bob has allowed it.) Meanwhile, some user-specific data is sensitive because it allows acting on someone else's behalf. (If Alice is logged-in to Amazon, Bob should not be able to buy things using her account.) Private and sensitive data are interchangeable from a systems perspective: they must be accessible by that user, and only by that user.

However, some data is only user-specific for convenience or organization; it isn' t important whether other users are able to read it, as long as it doesn't make their own actions less convenient.

For instance, the set of apps that are visible in menus might be one example of user-specific data that does not necessarily need to be treated as private. If Alice has installed apps for social media networks that Bob doesn't use, they shouldn' t appear in Bob's menus —but if Bob specifically looks for them, perhaps in an Android-Settings-like "storage usage"view, it might be considered acceptable that he can see what Alice installed.

Another possibility for sharing data is that playlists within a shared media library could appear as an unobtrusive "Bob's playlists"folder in other users' menus, if desired.

As discussed in Levels of protection between users, the level of privacy and integrity protection between users can vary according to OEM and consumer requirements; this could influence how user-specific data is categorized.

Authentication

We assume that the HMI provides a way for users to identify and authenticate themselves to a trusted HMI component, for instance by:

- presence of a unique physical key
- presence of a personal item such as a phone with Near-Field Communication support
- a password or lock-screen gesture
- face or fingerprint recognition
- simply selecting a user from a menu (choice of user, but no meaningful authentication, similar to one of the cases described in [Switchable profiles without privacy]

The exact authentication mechanism depends on manufacturer and user requirements, and is outside the scope of this document: this document only assumes that an identification/authentication mechanism exists as part of the operating system, and does not rely on specific properties of that mechanism.

General use-cases

While this document does not go into the specifics of more elaborate use-cases, there are a few simpler use-cases which should be considered by any concrete multi-user design within the framework established by this document. In some cases these use-cases could be considered and rejected, if a particular design's requirements put them out of scope. To show them, the context of a car will be used as an example.

First use

Alice uses the car for the first time. The system recognizes that she has not used it previously and so there is no saved state.

a. First use: The system starts in some default state, for instance at a main menu or with a default application such as a media player running.

Individual use: preferences and state restored

Alice and Bob share a car, and have separate keys. Alice has configured the display for a red UI theme; she uses the car on Monday, listens to a podcast while she drives, and has the email app open in the background. Bob has configured the UI for a blue theme. He uses the car on Tuesday, and reads the BBC News website in the browser app while stopped at motorway services.

a. Last-used mode: The next time Alice starts the car and authenticates as herself (see Authentication), the podcast and email apps should resume in the same state they were in when she shut the system down on Monday, and the HMI configuration should reflect her preferences (the red theme should be used, etc.). Similarly, the next time Bob authenticates as himself, the BBC News website should be displayed in the browser app as it was when he shut the system down on Tuesday, and the blue theme should be used.

b. Privacy between non-concurrent users: If the system is configured to provide protection between users, then Alice's private data should not be available to Bob and vice versa. For instance, Bob's web browsing history and social media accounts should not be available when Alice starts the web browser, even if Alice deliberately looks for them.

User switching

a. User switching: Bob is currently using the HMI to read Twitter, and Alice wants to check her email. Neither is currently driving. Alice should be able to authenticate in some way (see Authentication), switching the HMI to have Alice as its current user. When she has finished, Bob should be able to switch the HMI back so he is the current user again, and continue to read Twitter.

b. Privacy during user switching: after switching from Bob's user account to Alice's, Bob should be able to go away, knowing that Alice cannot access his Twitter feed. When Alice has finished and hands back control to Bob, she should be able to know that Bob cannot access her email.

> In existing multi-user systems like those described in section 4, this is typically implemented by leaving Bob's user account in a "locked" state after he transfers control to Alice, and vice versa, requiring re-authentication before resuming use.

Guest mode

Greg, a guest, is in Diana's car.

a. Unauthenticated guest session: If Diana has enabled it (or if it is enabled by default and Diana has not disabled it), Greg should be able to start a guest session that can access public information and the Web, play music from the car's music library, etc. without authentication.

b. Owner's privacy: Greg should not be able to access Diana's private data (or the private data of any other user of the system).

c. Guest's privacy: Greg's browser history, Facebook authentication token, etc. should not be available to subsequent guests. For instance, the system could temporarily allocate space for Greg's user-specific data, then discard it and terminate all guest processes as soon as Greg logs out, returning to default settings for the next guest.

d. Guest is restricted: Greg should not be able to add or delete music, install or remove apps, or similar actions.

Borrowing the car

Diana lends her car to David, giving him her key.

If the system is configured to consider a key as sufficient authentication for a user, then it cannot be expected to protect Diana from malicious action by David. However, if the system is configured to require secondary authentication such as a password, PIN or lock-screen swipe pattern, then David will not be able to use Diana's account.

a. Can create a new account: Even though David and Diana are using the same key, David should be able to create a new account that saves his preferences, and switch to it.

Existing multi-user models

This chapter describes the conceptual model, user experience and design elements used in various non-Apertis operating systems'support for multiple users, because it might be useful input for decision-making. Where available, it also provides some details of the implementations of features that seem particularly interesting or relevant.

Switchable profiles without privacy

The simplest multi-user model can be found in platforms such as Windows 98 and the Sony PlayStation 3. In these systems, certain settings and other pieces of application data (such as documents and saved games) are stored separately for each user, but there is no privacy or protection between users: each user can easily access other users'accounts.

One variant of this is where no authentication is required to access a different account, as on the PlayStation 3: a user selects their name from a list, and there is nothing preventing them from selecting a different user's name instead. Similarly, an unauthorized user can identify themselves as any authorized user and gain access.

Another variant of this is where there is meaningful authentication (e.g. a login step with a password), but authenticating as *any* user is sufficient to access *all* users'private files. For instance, Windows 95 offered login authentication, but did not support filesystems with user-level permissions. As a result, unauthorized users were prevented in principle (in practice, the login step was easily circumvented), but each authorized user had the technical capability to read and write any other user's files by navigating to the appropriate directory.

Both variants of this model are simple to implement, and provides straightforward semantics. Their disadvantage is that they do not meet typical privacy expectations for a modern operating system: users can impersonate one another, read each other's private files, and even alter each other's private files. As such, it is only suitable for an environment in which every user of the system fully trusts every other user of the system (and, for the first variant, everyone with physical access to the system).

We anticipate that these simple use-cases will be appropriate for some, but not all, Apertis systems: for example, they might be appropriate for a family car where the installed apps do not handle particularly sensitive information. In other Apertis systems, stronger privacy/protection between users is likely to be required.

Typical desktop multi-user

Many modern desktop/laptop operating systems (such as Windows 10, Mac OS X, and various open source desktop environments on Linux and BSD platforms) have a similar model for how multiple users are handled. Apertis shares many software components with the GNOME 3 desktop environment (as used in, for instance, Debian GNU/Linux and Fedora Linux), so we will use GNOME on Linux as our primary example of this type of environment.

On Unix-derived systems such as Linux and Mac OS X, each user account is typically represented by one Unix uid, corresponding to their intended use in all Unix systems.

Basic multi-user: log out, log in as another user

The most basic form of multi-user support is considerably older than graphical user interfaces, and is implemented in most current desktop/laptop operating systems. The system boots to a login prompt at which the user can choose their user account (for instance by choosing from a list or by typing its name), and authenticate in some way (typically with a password, but many authentication mechanisms are possible).

Each user has their own set of data files and configuration. To provide privacy between user accounts, the system tracks the ownership of user files, and either denies access to other users'files by default, or can be configured to do so.

To switch between users, the first user must log out, ending their session; this typically also terminates most or all of their user services. Ending their session presents another login prompt, at which the second user can log in.

In a typical implementation on Linux systems with the X11 windowing system, a system service (a "display manager", such as GNOME's GDM) starts an X display and uses it to show the graphical login prompt. When the first user logs in, their uid is granted access to the X display, which is taken over by their session. At the end of their session, the display manager terminates the X server, and starts a new X server for the next login prompt.

Systems which offer this model can easily support the simpler models from [Switchable profiles without privacy] as trivial cases of this model: they can implement the PlayStation 3-like model by omitting the authentication step after choosing a user, or the Windows 95-like model by giving each authorized user access permissions for other users'files.

Fast user switching: switching user without logging out

A refinement of the above model for systems with enough memory is to offer more than one parallel login session, with one active login session and any number of inactive sessions. This is commonly referred to as *fast user switching*.

Again, most current desktop/laptop operating systems offer this in some form. The first user chooses a "Switch User⋯"option from a menu; this optionally locks the first user's session (for instance by locking their screensaver), and switches to a login prompt at which the second user can log in. To switch back, the second user uses "Switch User⋯"to access another login prompt, at which a third user can log in, and so on. Several users can share the system, with up to one active session and any number of inactive sessions (limited by system RAM, and optionally an arbitrary limit on the number of users).

If the user logging in at the login prompt already has a login session, then the system detects that, and instead of starting a new session, it switches back to the existing session, automatically unlocking the screensaver if required. When a user logs out, their session is replaced by a login prompt at which any user can log in.

Designers typically treat this model as a superset of the simpler model in Basic multi-user: log out, log in as another user: in practice, implementations of fast user switching also offer the non-concurrent log-out/log-in arrangement as a trivial case. Similarly, as in Basic multi-user: log out, log in as another user, implementations of this model can easily support the models from [Switchable profiles without privacy] as trivial cases.

In GNOME's GDM display manager, the first session takes over the X server originally used for the login prompt, the same as in Basic multi-user: log out, log in as another user, this runs on a Linux virtual console, traditionally tty7. The "Switch User⋯"option causes the display manager to run a new X server on a different virtual console, typically tty8, and switch to it; the second user' s session takes over that X server, and so on, allocating a new virtual console and running a new X server each time. If a user logs out, the display manager remains on the same virtual console, but runs a new X server for the login prompt. If the user logging in at the login prompt already has a login session, instead of taking over that X server for a new session, the display manager switches to the appropriate virtual console for the existing session. The X server with the login prompt remains in the background, and is re-used the next time a login prompt is required, instead of starting a new X server: for example, a system where three users Alice, Bob and Chris repeatedly switch between their accounts would reach a "steady state"with four X servers on four virtual consoles (corresponding to Alice, Bob, Chris, and the login prompt).

Once two or more users have logged in, this model provides very rapid switching between them: none of their applications or user services need to be terminated or restarted. It also eliminates any loss of transient “context”such as notifications or window positions, without needing to implement state-saving. However, it uses a significant amount of memory: because inactive users’applications are not terminated, two alternating users could need up to twice as much memory as a single user. Similarly, because the inactive users’applications are not terminated or paused, merely disconnected from input and display devices, they can continue to consume other resources, such as CPU time and network bandwidth: a misbehaving application in Alice’s session can cause Bob’s session to appear slow.

Multi-user desktops with multi-seat support

Some systems, in particular the systemd-logind component used in Apertis, can be used to extend the model in Basic multi-user: log out, log in as another user by offering several so-called “seats”as defined in Multi-seat logind seats. A logind seat is a collection of display and input devices intended to be used by a single user, offering the equivalent of section Basic multi-user: log out, log in as another user independently on each logind seat. Similarly, a system can offer “fast user switching”(Fast user switching: switching user without logging out on some or all of the available logind seats.

GNOME’s GDM display manager switches between virtual consoles on the first logind seat, in exactly the same way as section Fast user switching: switching user without logging out. On the second and subsequent logind seats, it behaves as described in Basic multi-user: log out, log in as another user, with this logind seat’s X server remaining visible regardless of the current virtual console, and does not offer “fast user switching”.

Android 4.2+

Recent versions of Android have gained multi-user support, initially for tablets only, then extended to phones in Android 5.

When first started, Android 4.2[1] shows a prompt for setting up the first user account. The first user account is special in that it is considered the administrator for the device, and can thus create, remove and assign permissions to other users.

Android uses separate Unix user account IDs (uids) for separating applications from each other, so any communication or sharing between applications was already mediated by the Linux kernel and other trusted parts of the Android system software. The multi-user design simply allocates a block of uids to each user, one uid per (user, application) pair: for example, the first user (user

[1] http://developer.android.com/about/versions/jelly-bean.html#android-42

number 0) might receive uids u0a123 and u0a45 for two of their apps, and user number 1 might receive uids that include u1a67.

Because applications are already isolated from one another by their differing uids, all interaction between apps is mediated by trusted processes, so those trusted processes were adapted to take the user into account when deciding permissions. Similarly, because apps conventionally use Android-specific APIs to access user data, adapting those Android-specific APIs to take the user into account is straightforward: an application making an API call that previously listed *all* online service accounts will now only be told about the appropriate user's online service accounts.

Authentication is through the usual means used by Android: each user gets their custom lock screen and, depending on that user's settings, types in a PIN, a password or a pattern connecting dots in a grid for logging in. Icons representing all users are shown in the current user's lock screen, so user switching is a matter of locking the screen (which can be done through the 'quick settings' menu, available in the status bar) and tapping the desired user.

From a user interface perspective, this resembles Fast user switching: switching user without logging out on typical desktop operating systems. However, as an implementation detail, each user's apps are terminated when user switching occurs, so the actual implementation is closer to the "log out / log back in"model (section Basic multi-user: log out, log in as another user).

Some settings are global to the device, including Wi-Fi networks. All users can change these settings, apparently, and those changes will affect every other user. User settings and data are kept separate from each other's. The list of applications in the user's launcher is separate for each user, but application files are only downloaded the first time a user asks that application to be installed, to save space.

Because Android provides custom API for everything the application does, the storage and reading of data and settings for each user is done automatically by their APIs. That means applications did not have to be modified for supporting multi-user: the fact that they already use Android APIs to obtain directory paths and save files ensures that they are saved to the proper place.

Multi-user support in the Tizen 3 automotive platform

The multi-user architecture designed for [Tizen 3] in an automotive environment was presented at FOSDEM 2015.

At a conceptual level, Tizen applications can either be installed system-wide or for a particular user. Guest users can only use system-wide applications; it was not clear from the presentation whether only preinstalled applications can be system-wide, or whether separate installable applications can also be installed system-wide. If installed for a particular user, the application's files are copied into that user's home directory, contrasting with the centralized app storage used “behind the scenes”in this design document and in Android.

The Tizen model is designed for a “multi-seat”environment as described in Multi-user desktops with multi-seat support, where several sets of grouped devices (a display, its attached touchscreen input device, and perhaps USB sockets and/or a headphone jack located near that display) are all attached to the same computer as peripherals; this is an attractive model if the system is powerful enough to provide acceptable performance on all seats, but comes with higher performance requirements than some of the potential classes of requirements addressed by this document. In particular, there is a focus on the ability to move concurrent applications seamlessly from one screen to another, following a user who moves from one seat to another.

In the Tizen model, all users share a single compositor, which manages all seats' displays and input devices, resulting in the compositor being required to act as part of the TCB for security between users (see [Trusted components]). As discussed further in Graphical user interface and input, we do not recommend this approach while using X11 for GUI services.

There is a single privileged user in the Tizen system, and only that user can configure certain shared resources such as wireless networking and Bluetooth. This seems an unnecessarily limiting model for a car that might be shared between two or more primary drivers, for example in a family. It is intended that this user will eventually be able to launch applications on seats that are currently in use by other users.

The API model in Tizen appears to involve system services such as the media server and thumbnail generation service not only acting on behalf of users to fulfill requests, but running as ‘root’so that the same application can write directly into multiple users'home directories. We recommend avoiding this practice: it puts all of those services into the TCB for each layer of the security model (security between users, security between apps and security between system services), greatly increasing the amount of security-sensitive code in the system and the potential impact of a bug or security flaw.

The presentation mentioned adding the user ID as an explicit parameter in IPC (inter-process communication) calls from applications to system services so that the system service will act on behalf of the appropriate user. This could be made to work securely by verifying that the actual user ID matches the one in the IPC call, but is a potentially dangerous approach: if a naive implementation trusts the given parameter and does not verify it, a malicious application could easily subvert that implementation. We recommend avoiding “user ID”parameters in APIs: if the service can determine the user ID in a secure way, then the parameter is unnecessary, and if it cannot, this approach brings the calling application into the TCB for security between users (with the practical result that all or nearly all applications would end up in the TCB, greatly increasing the system's attack surface).

Approach

Because this document does not define precise requirements or use-cases for the system, this section outlines multiple possible approaches to several design questions. The choice between these approaches must be made based on concrete requirements.

The principle of least-astonishment

One valuable general design principle is that, when a user carries out an action, it should be easy to predict the outcome. In the context of a multi-user system, this implies various more concrete principles, such

- sharing should not occur when a user would not expect it to; this "oversharing"is likely to lead to users distrusting the system and being unwilling to store private data in it, even if that would be advantageous
- sharing should occur when a user would expect it to; if it does not, users will be inconvenienced by having to copy data manually between different contexts
- performing a similar action in different contexts should have a similar result

Levels of protection between users

There is a spectrum of possible sets of requirements for privacy and integrity protection between users: a strongly protected model similar to the one detailed in section [Typical desktop multi-user], a model with no protection at all as described in [Switchable profiles without privacy], or anything in between (e.g. with protection between users in general, but certain categories of data explicitly shared).

The desired level of protection depends on the user, but we could also decide that Apertis will only support a subset of the possible range, and an OEM could decide that they will only support a subset of the range allowed by Apertis.

In use-cases that involve differently-privileged users, the desired level of protection might vary between users within a system: for instance, the main users of a car might opt for a setup in which switching from one main user to another does not require authentication, but switching from a "guest"user to a main user does.

For each set of requirements, we aim to minimize the "friction"in switching between users, subject to whatever minimum is imposed by the requirements – stronger privacy and integrity protection comes with a higher minimum "friction". For example, if users are to be protected from each other, then switching between users must include an authentication step, whereas if there is no effective protection (privilege boundary) between users, switching between users merely requires choosing the desired user account.

As a general design principle, design documents for concrete use cases should address the “strongest”supported protection between users, because that imposes the most difficult privacy/integrity requirements. Secondarily, they should consider the “weakest”supported protection between users, because that imposes the most general sharing requirements: ideally, this is just a trivial case of the high-privacy version, with some of the “pain points”omitted, but it does introduce new requirements for the ability to pass data between users. All other levels of privacy/integrity protection can be represented as somewhere between those extremes.

As a compromise plan if we find situations that cannot be solved in a higher-privacy model, it is possible to relax our requirements to declare the highest-privacy use cases to be out of scope.

User accounts: representing users within the system

There are three possible approaches to representing users in a Linux system.

Sharing one uid between all users

In this approach, all user applications and user services run under the same uid. The system defines its own proprietary “user account”concept, and all components that access user-specific data must ensure that they access the correct user’s data, disallowing access to other users’data if appropriate.

This has the potential to make transitions between users very easy: the “current user”is simply a variable within each application or service. However, it places a great deal of trust on each of these components, including every third-party (user-installable) application that accesses user-specific data. If the system’s security model is that users can be protected from each other, then in effect, all of these components are included in the trusted computing base; if the requirements do not include protection between users, then distinguishing between users is not required for security, but is still required for correctness. In practice, we anticipate that not every component would discriminate between users correctly.

This approach also has practical problems for the re-use of existing open source components, which assume the traditional use of one uid per user. Having to modify all of these components, with a complex change that is unlikely to be accepted by their upstream developers, would significantly reduce the competitive advantage derived from their use.

As a result of these disadvantages, we do not recommend this approach for Apertis. It would only be viable if all of the following are true:

- users are not protected from each other, and this will not change in future development
- user-specific data is minimal, only needs to be accessed via Apertis-specific APIs, and this will not change in future development
- it is not considered to be a significant problem if third-party applications and services do not consistently distinguish between users, and this will not change in future development

An additional consideration for this approach is that it potentially alters a large number of interfaces (such as D-Bus method calls) to have a parameter for the user account to be affected. If changing requirements result in switching to the “one uid per user”or “many uids per user”models in future, such that the correct user account is implicit in the uid, then this vestigial parameter will remain in the interface, making the interface more complex than is required.

If the form of the additional parameter resembles the numeric or string form of a uid, then this could even lead to security issues, for instance if a component trusts the explicit user-account parameter and ignores the actual uid.

If this approach is taken, then we recommend reducing the confusion caused by naming the additional parameter something more similar to “profile”than “user”. If the system is later extended to have one uid per user, rendering the parameter vestigial, we recommend giving it a neutral, constant value that does not match any user account name, such as “default”.

One uid per user

The traditional Unix design which motivated the uid concept is that each user account is represented by one numeric uid.

Because each process (i.e. each application or service) starts with a particular uid, and processes without administrative privileges cannot change their uid while running, this approach requires that user-switching involves starting new processes for the new user.

The major advantage of this approach is that it is how the existing components in the system, including the Linux kernel, are designed to operate. In particular, the Linux kernel provides privacy and integrity protection between uids.

We recommend this approach for Apertis.

Multiple uids per user

Android uses a design involving multiple uids per user, one per app or set of related apps, as described in Android 4.2+. This allows the Linux kernel’s privacy and integrity features to be used to protect apps from other apps, even within a user session. However, in Apertis, this advantage is redundant, since we already use a different kernel feature (AppArmor) to provide privacy and integrity protection between apps.

The major disadvantage of this approach is that it requires every interaction between dissimilar apps to be mediated by a system-level component. Within the context of Android, this is not a problem, since Android applications and services are expected to use Android-specific APIs in any case. However, Apertis re-uses existing open source components where appropriate; these components would have to be modified to cope with crossing privilege boundaries when they communicate with different uids, which, as in the "one shared uid"approach, would reduce the value of re-using these components.

We do not recommend this approach for Apertis.

Creating and managing user accounts

Based on the description of desired use case scenarios, Apertis understands the main means of identifying and authenticating a user will be through a external medium, such as a car key.

If runtime user-switching is required, a secondary form of authentication is likely to be required. This could be done via a password (or equivalent, such as a PIN or touchscreen swipe pattern), via biometrics such as fingerprint, face or voice recognition, or by verifying possession of a near-field communication device such as a mobile phone.

As previously noted, depending on manufacturer and consumer requirements, there is the possibility of simpler authentication schemes for less privacy-conscious users; for instance, a manufacturer or consumer could choose to relax the security model to one where the external medium is sufficient to authenticate as any registered user selected from a menu.

A registration process will be required, to associate authentication tokens with user accounts: one way this could work is detailed in this section.

Registering the users

When the system in run for first time each user in turn will follow the following procedure:

1. A user is authenticated by an external medium, such as a car key
2. The Apertis system starts up and recognizes that it is a new unregistered user
3. A wizard is displayed to register the new user
4. The user enters whatever information is needed to set up their user account, such as their name

5. The user is given the option of registering a password or other authentication tokens to be used for keyless authentication (for user switching, mainly)

6. Alternatively the wizard can continue from here on to register email and web accounts the user may be interested in

In case there are more users than keys available, new keys will need to be acquired.

The first user to be registered is special

It's important that at least one user be able to perform administrative tasks, such as wiping out all of the data, removing users, and so on. One practical solution to this is that the first user to be registered is considered special and be able to perform these tasks and is also able to give these privileges to other users as they see fit, so that more users would be able to perform administrative tasks.

One analogy used in the security literature is that the system "imprints"on the first user seen, in the same way that a duckling imprints on its parent. A refinement of this model is that deleting all users resets the system to a state in which the next user created will be privileged, the so-called "Resurrecting duckling[2]"model.

> Frank Stajano and Ross Anderson. *The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks.* In B. Christianson, B. Crispo and M. Roe (Eds.). *Security Protocols, 7th International Workshop Proceedings*, Lecture Notes in Computer Science, 1999.

Graphical user interface and input

This section explores several potential models for managing input and output.

The basic infrastructure component for Wayland is a *compositor*, which is responsible for mapping application-supplied surfaces (windows) into the visible display, routing input events to those surfaces, and applying any visual effect with a larger scope than an individual application, such as animated transitions between applications.

The Wayland compositor is part of the TCB for security between apps: it is responsible for imposing a boundary between the apps that communicate with it, and preventing them from carrying out undesired actions such as reading each other's input or taking screenshots of each other's windows. Depending on the design and implementation, it may also need to be part of the TCB for security between users.

[2] https://www.cl.cam.ac.uk/~fms27/duckling/

In the current design, the compositor being used is agl-compositor. It is a Wayland compositor based on `libweston`.

Single compositor

One possible model is to have a single compositor which starts on boot, runs until shutdown, and is directly responsible for compositing all application surfaces. This model would be appropriate if there is only one uid shared by all users as described in section Sharing one uid between all users, since in that model there is no OS-level isolation between user accounts in any case. It could potentially also be used in a design where each user has their own uid, by running the compositor with a non-user-specific uid.

The major disadvantage of this situation is that it places the user-level compositor into a trusted position: it would become part of the trusted computing base for separation between users (see [Trusted components]).

As a general design principle, the less code is in the trusted computing base (for any given layer of security),

the better; this conflicts with the user-level compositor's broad role in mediating between apps, including animated transitions, copy/paste functionality, on-screen keyboard handling and so on.

Switching between compositors

The traditional design for user-switching in X, as described in Basic multi-user: log out, log in as another user and Fast user switching: switching user without logging out, is to start a new X server for each user session and switch between them, for instance by using the Linux kernel's “virtual console”facility, or by dynamically attaching/detaching the X servers to the video device.

A possible implementation for this scenario is to run multiple session compositors, switching access to the video output between them, and not having a system compositor.

In this model, the transition between users would involve systemd-logind revoking the old session compositor's control over the display (“DRM master”status) and over input devices, and giving control to the new session compositor. This could be done at any point in the transition: before, after or during an animated transition.

The major disadvantage of this design is that switching between virtual consoles is an all-or-nothing operation: the system either displays a frame from one compositor or a frame from another, but it cannot combine two (for instance by overlaying them, with transparent regions). It is also not instantaneous, and would have to be disguised by having a transition where several consecutive frames are allowed to be the same.

For some UX designs, this would not matter. For example, if a designer specifies that the first user's session should "fade out"to a black screen or some sort of "please wait⋯"placeholder, or move off-screen, then the system could switch to a matching frame in the new compositor, wait for the switch to occur, and have the second user's session "fade in"or move in from off-screen. Similarly, if the UX for user-switching involves a menu from which the new user is chosen, then that menu could be used as a fixed point around which to anchor the transition.

However, if the desired transition has the two users'sessions overlap –for instance, a full-screen cross-fade from one to the other, or any animated movement that has both sessions exist on-screen at the same time –then it would be difficult to achieve these effects in this design without essentially copying a static screen-capture of one session into the other session. Similarly, if the desired transition has smooth movement from beginning to end –for example, smooth horizontal scrolling with the conceptual model that the other user's session is "just off-screen"–then the only practical points at which to do the virtual console switch would be at the very beginning or at the very end; either way, this would likely result in a few frames of non-responsiveness at a time when the user might reasonably expect the system to be responsive.

Copying a screen-capture of one session into the other session is also a potential privacy risk, since it results in the screen contents crossing the trust boundary: it would be technically possible for the second user's session to save the captured image.

Switching between compositors with a system compositor

Because Wayland does not require clearing the framebuffer during switching, another possible approach would be to use a system-level compositor without nesting, used for transitions, and optionally for startup and shutdown. At any given moment, either the system-level compositor or a session compositor would be active (have control over input and output), but never both.

In this model, as in [Switching between compositors], the transition between users would involve systemd-logind revoking the old session compositor's control over the display ("DRM master"status) and over input devices; however, instead of immediately giving control to the other session, instead it would give control to a special-purpose system-level compositor which would perform the transition, and then in turn hand over to the new session. This system-level compositor could capture the current screen contents as a starting point for the animated transition, if desired; as in [Switching between compositors], the screen contents would cross a privilege boundary, but unlike [Switching between compositors], the other side of the privilege boundary in this design is a trusted process.

The new session compositor could be started without direct access to the display (it would not yet be the "DRM master"), and instructed to draw its initial state into a buffer; recent Linux kernel enhancements mean that it could use in-GPU processing and memory for this drawing operation, without having control over what is displayed. The system-level compositor would use that buffer as the endpoint of its animated transition. On completing the transition, it would instruct systemd-logind to grant full display and input access to the new session compositor.

As a result of its role in user-switching, the system-level compositor used for the transition would potentially be part of the TCB for security between users. However, its functionality would be minimal: because it would not be active during normal use, only during transitions, it would not necessarily need to process input at all, and its output handling would be limited to performing the animation from the old to the new screen contents.

Switching between users

If runtime switching between users is required, there is a spectrum of possible approaches.

At one extreme is the simplest form of the approach described in section 4.2.1, where we terminate all of the newly inactive user's apps and user services (anything that is user-specific), and only non-user-specific processes (system services) continue to run. That has the lowest possible memory and CPU overhead: there is going to be a small amount of overhead during the necessary "grace period" while we let the inactive user's apps save their state before killing them, but this is minimized.

At the opposite extreme is the "fast user switching"as described in section 4.2.2, in which the inactive user's entire session, including GUI apps, user services, games, and infrastructure components such as the window manager and session compositor continue to run, with the only difference being that they are disconnected from the input and display hardware. That has considerable overhead: in the worst case, where we assume that system services are negligible when compared with per-user components, switching between two users could double the memory and CPU consumption.

We can choose various points along that spectrum depending on OEM and customer requirements. If we can terminate all of the inactive user's apps and the majority of their user services, the result is close to the first extreme –for example, this could be based on an "agents continue to run across user-switching" flag in the app manifest, perhaps implemented as an Android-style "permission" . App-store curators could carry out more thorough validation on services that request that flag, to ensure that they will not have an adverse performance impact.

If we can terminate all of their apps but must leave *all* of their user services running, we get closer to the second extreme. The closer we are to the second extreme, the higher our hardware requirements for a given performance level will be.

If we terminate at least some of the newly inactive user's processes, a second axis of variation is how much overlap we are prepared to tolerate between the sessions: to allow those processes to save their current state, a “grace period”will be required between notifying those processes that they must exit, and actually terminating them.

One approach is to disallow overlap entirely, and not start the transition until the inactive user’s session has completely ended, with a “please wait⋯”message while their processes shut down. However, this maximizes latency and user-visible disruption. To reduce the time required to switch between users, it might be desirable for these processes to continue to run concurrently for a short time, in parallel with starting the newly active user’s session. There is a trade-off here: the more CPU time is consumed by the newly inactive user’s processes, the less is available to display a smooth animated transition to the newly active user and launch *their* processes. This could be mitigated by de-prioritizing the CPU and bandwidth consumption of the inactive user’s apps, at the cost of extending the necessary “grace period”for a given amount of state-saving activity: for example, if an app’s state-saving procedure would normally take 50% of the CPU for 0.1 seconds, throttling that app to 5% of the CPU would make its shutdown take 1 second.

Preserving “core”functionality across user-switching

If user-switching during use is supported, then certain features of the system must continue to work during and after the user switching operation.

For example, navigation-related notifications (notifying the driver that they should turn off their current route soon, that the speed limit will change soon, etc.) are time-sensitive, and it would be reasonable to require that these notifications are not interrupted or delayed, even if user switching takes place just before or even during the notification.

Further examples of background features that might be in the category that must not be interrupted include media playback (if the driver is listening to music, it would be reasonable to require that playback is not stopped or disrupted by user switching, although interrupting “now playing⋯”notifications might still be acceptable) and incoming phone or VoIP calls.

These features cannot be assumed to be a fixed part of the operating system: for example, it should be possible to have uninterrupted media playback via a third-party audio streaming app, such as one for last.fm or Spotify, or uninterrupted VoIP call notifications for a third-party VoIP implementation.

Conversely, essential operating system features such as preinstalled or non-removable apps are not necessarily all in the category of features that must continue to work during user-switching: for example, incoming email notifications are less time-critical than calls, and it is likely to be acceptable for them to be paused during user-switching.

There are several possible approaches to keeping these features working across a user-switch. Depending on the concrete requirements and use cases, we could choose one of these approaches for the whole system, or choose some combination of them for different apps and services.

As mentioned briefly above, there is the potential for a subtle distinction between components where an interruption to notifications is unacceptable (for instance, navigation or incoming calls might be in this category), and components where an interruption to functionality is unacceptable, but an interruption to notifications is allowed (or even desirable).

For a possible example of the second category, consider music playback, on a system where a visual notification is triggered when the current track changes. Suppose we switch the current user from Alice to Bob at 12:00:00, at which time track 1 is 2 seconds from ending, and the animated transition takes 4 seconds. It seems reasonable to expect that track 1 must continue to play until 12:00:02, and it also seems reasonable to expect that track 2 must start at 12:00:02 and continue to play smoothly. However, it is not necessarily a requirement that the “now playing track 2”notification cannot be delayed until Bob’s session becomes fully available at 12:00:04; indeed, this might be considered more desirable than having it interrupt the animated transition.

System services

System services (as defined by [System services]) continue to run regardless of what is happening in user sessions, so one possible approach is to put “core” functionality in system services. These could be anywhere from highly privileged to entirely unprivileged; the distinction here is only that they are independent of user accounts.

For example, network management services such as ConnMan are highly-privileged system services, whereas the Avahi name-resolution and service discovery service is system-wide but unprivileged.

If this approach is to be used for third-party installable applications, then we will need to ensure that third-party application bundles can provide system services, in a way that does not allow those third-party application bundles to compromise the overall security of the system.

For components that deal with user-specific data, making the component into a system service requires that the component is trusted to provide the correct privilege separation: for example, if the component has access to multiple users’ private data, it should not reveal one user’s private data to another user unless the system’s security model allows this to happen.

As a general design principle to avoid circular dependencies and unnecessarily tightly-coupled components, lower layers should not rely on higher layers. System services are at a low layer in the stack, so they should not initiate communication with user services or users’graphical sessions. One common approach to this is to have a component inside each user session whose role is to provide the user interface for a “headless”system service, separating backend logic and system-level configuration (the system service) from user interface presentation and per-user configuration (the user part).

User services continuing to run

User services (as defined by [User services] are inherently per-user. If the end of a user’s login session terminates their GUI applications but leaves some or all of their user services running, this could increase system load (as noted in section [Switching between users]), but would make user services a suitable implementation for features that must run uninterrupted. This could apply either in general, or with restrictions (for example, some subset of the inactive user’s user-services could continue to run, perhaps according to a “flag”in their associated app manifests).

Distinguishing between the driver and other users

Because the driver is the primary user of the system, one possible refinement of this requirement would be to say that core functionality associated with the driver cannot be interrupted, and must retain its ability to display notifications, but that switching may interrupt functionality associated with other users. This would limit the additional system load from multiple users: the maximum set of processes running at a given time would be one non-driver’s full session, plus whatever subset of the driver’s processes are considered to be necessary.

Agents

The Apertis design has the concept of “agents”, which are lightweight background processes running on behalf of a user. Depending on the precise requirements for agents, they could be implemented as system services, or as user services, or divided between those two categories.

Returning to previous state

Saving and restoring the state of the session is a hard problem in general. Some platforms, such as Android, made it a central piece of their application life cycle management and built it right into the application support for the platform. The fact that Android and iOS have custom platform layers allows them to make this viable.

Apertis is not aware of any deployment of OS-level freezing and thawing of processes at the moment, but such a strategy could be investigated in the future for usage in Apertis. For now, having the application itself care about saving and restoring state, even if supported by some high level API, seems to be the more realistic approach. More discussion about this can be found in the Applications design[3] document.

[3] https://www.apertis.org/concepts/archive/application/applications/

Application ownership and installation

In current app-store platforms such as Apple, Google Play, Steam or PlayStation Store, if you buy an app, it is associated with your personal account (Apple, Google, etc.) and can be downloaded to any device associated with that account, subject to some limits. This is one possible approach to how apps are deployed on Apertis.

To avoid wasting space with duplicate application installations, current app-store implementations with multi-user support, such as Android, have chosen to install applications system-wide. If Apertis apps are, conceptually, installed per-user, then we recommend implementing this by keeping a list of apps per user, and merely hiding apps from users who have not “installed”that app. If the user acquires an app that another user has already installed, the system could behave as though it was freshly downloaded, but in fact just stop hiding the system-wide app from the current user: from the user’s perspective, this is indistinguishable from a very fast download and installation.

Another potential conceptual model is to treat apps as more like car accessories. You could, for instance, buy a car with metallic paint, or add alloy wheels later; when you sell the car, the feature goes with it. Applying this model to applications, it could be possible to buy a car with the social media app bundle preinstalled, or add the media streaming bundle later, and have the apps go with the car when it is sold. In some respects, this is the more natural model from the implementation point of view: we do not recommend duplicating the app’s executable code and resources, regardless of whether it is conceptually installed per-user.

Whichever of these approaches is taken, choosing whether ownership/licensing of the app follows the car or the purchaser is primarily a matter for the app store implementation, not the multi-user design.

Summary of recommendations

As discussed in [User accounts representing users within the system], Apertis recommends representing each user account as a Unix user ID (uid). The first user to be registered in a new system must be able to perform administration tasks such as system updates, application installation, creation of new users and setting up permissions –that is discussed in Creating and managing user accounts.

There is a range of possible approaches to switching between users, discussed in section [Switching between users]. This document does not recommend a particular choice from that range, since it depends on the available hardware resources and the system’s use-cases and requirements. For budget-limited designs with significant hardware limitations, we should consider terminating most user-level processes while switching to reduce concurrency, or if this is not acceptable, opt to leave user-switching unsupported; for premium models with more capable hardware, the more resource-expensive "fast user switching"approach can be considered.

In Preserving "core"functionality across user-switching we outline various possible approaches to ensuring that "core functionality"is not interrupted by a user switch. Services that need to stay running after a user switch should have their background functionality split from their UIs; they can either run as a different Unix user account ID –a "system service"–or be a specially flagged "user service" that is not terminated with the rest of the session.

In Returning to previous state, Apertis recommends that applications should be handling saving and restoring of their state themselves, potentially supported by helper SDK APIs, which means only applications written with Apertis in mind would work. That recommendation comes from the fact that there is no solution that would work for all applications.

Ways of having a smooth visual transition when switching users are discussed in section Graphical user interface and input. This should be implemented according to OEMs needs.

Switchable profiles without privacy) [Switching between compositors]: #switching-between-compositors [Switching between compositors with a system compositor]: #switching-between-compositors-with-a-system-compositor [Switching between users]: #switching-between-users [System services]: #system-services [Tizen 3]: https://fosdem.org/2015/schedule/event/embedded_multiuser/ [Trusted components]: #trusted-components [Typical desktop multi-user]: #typical-desktop-multi-user [User accounts representing users within the system]: #user-accounts-representing-users-within-the-system [User services]: #user-services