emlix
embedded linux systems

White Paper & Case Studies

Industrial Android Security: System Security throughout the Product Life Cycle

by Martin Homuth

Summary

Android offers flexibility and usability that is unmatched and these benefits have been recognized for use in industrial systems. This paper shows that industrial devices based on Android do provide certain challenges, but the re-architecture of Project Treble and pursuing the necessary approach during system development offer chances to create long-lasting products that can be updated and secured for longer periods than previous or current consumer products. The experience of customers shows that once the product progression leans towards separation of the Android framework the complexity of updates decreases significantly.

Table of Contents

emlix
embedded linux systems

Industrial Android Security

System Security throughout the Product Life Cycle

by Martin Homuth

Introduction

Android is an open-source software stack building upon a Linux kernel, which in addition to smartphones and tablets is increasingly employed in industrial products. Besides the free availability of Android itself, the comprehensive infrastructure for app development and the usability experience already established worldwide are important motivational factors for the expansion into the industrial domain. Furthermore, Android as a closed-community project is continuously being improved and enhanced. Security-related adjustments are available to vendors regularly and within short cycles.

Industrial products based on Android differ greatly from typical consumer devices. For such devices the functionality and hardware interfaces are fundamentally predefined by Google and supported in their code base. Industrial devices incorporate very different hardware components. Communication with a microcontroller for the control of custom hardware or the integration of a field bus interface are just two examples of the numerous possible requirements that occur in the industrial domain. Furthermore, the quantity of devices is far lower and the user base of the system is fairly narrow with a more restricted nature of system usage.

This paper describes the architecture of an Android system and its security mechanisms and focuses on the peculiarities of porting and updating Android systems for industrial products. Consequently, the party of interest is the OEM vendor, the entity that assembles products based on SOC vendor components to be sold by other manufacturers later. The particular focus is on the re-architecture introduced with Project Treble, which is the foundation for a distinct separation of the platform and the Android framework and as such is of utmost importance for industrial products with long life cycles that need to be kept up to date. A cost and time-efficient deployment of Android systems can only be achieved by introducing this separation, as the vendor is then able to identify and evaluate the complexity drivers at first hand. Additionally, when updating devices the vendor is now in a much better position to decide selectively which components to update and which to leave unmodified without compromising the functionality and security of the system.

Based on a case study, the development of a new industrial product with the use of recent Android security mechanisms is outlined. After porting and a customization of the AOSP (Android Open Source Project) to product-specific hardware, it may be required to integrate additional peripheral components. Subsequently, the security mechanisms can be adjusted to the concrete use cases of the product, without compromising the integrity of the system.

In another typical example it is shown how the security of inventory products in the field based on different versions of the AOSP can be preserved and improved. The possible risk and complexity factors that may occur during the implementation are identified and explored. Moreover, the mechanisms used to distribute and deploy system updates are outlined.

During the course of this paper, a non-standardized nomenclature needs to be defined for Android systems before and after Project Treble, as only the changes introduced through this architectural re-engineering allow for cost-efficient and flexible system development that is able to deliver high quality on the part of the vendor and Google in a fraction of the time. We will differentiate these system classes into legacy Android systems (Android version 7 and earlier) and modern Android systems (Android 8+)

The chapter „Android Open Source Project (AOSP)“ introduces the Android Open Source Project and describes in subsequent sections the core components and concepts of Android systems to allow the reader to obtain a basic impression of the fundamental requirements. The chapter „Android Security Features“ introduces the core security features of Android systems that may be implemented in products, followed by the chapter „Porting Android to Industrial Devices“, which describes the process of porting Android systems to new hardware. The chapter „Security and Feature Updates of Android Systems“ illustrates the details of performing system and security updates. The chapters „Case Study: Porting of a Secure Public Transport Ticketing System“ and „Case Study: Updating Legacy Android Systems and its Challenges“ conclude with the two aforementioned examples, describing the processes of updating and porting industrial Android systems respectively.

Android Framework

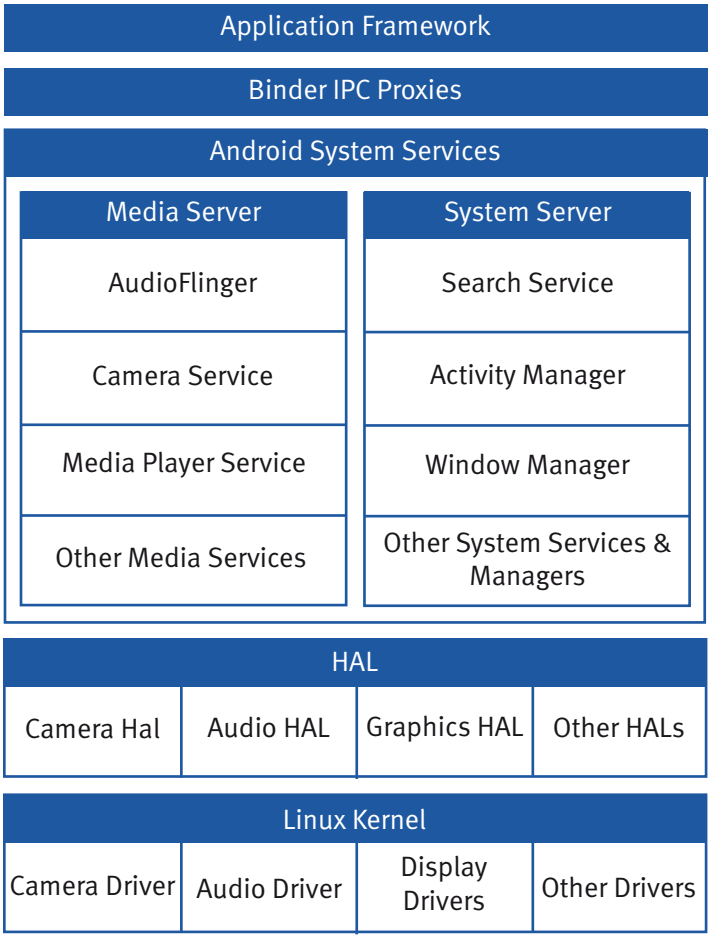


Figure 1: Schematic illustration of Android architecture

Android Open Source Project (AOSP)

The Android system is fundamentally distributed through the Android Open Source Project (AOSP) and is publicly and freely available for anyone interested. Components that are licence-constrained such as the Google Mobile Services [1] are not part of the public repository. Additionally, even though the Android kernel is open source, it too is not part of the AOSP tree, as it is GPL-licensed and Google works very hard to keep the contents of the AOSP under very liberal licences. The reason for this is not disclosed but it might allow Google to close up the system at some point in the future. The repository itself, however, contains the full Android stack required to build a fully fledged Android system.[1] Figure 1 illustrates the architecture of an Android system.

1 Part of the AOSP tree is the different flavours of Android, such as Automotive, TV or even core components used in Fuchsia

Fundamentally, Android is a highly customized embedded Linux system that utilizes the Linux kernel for basic resource management (hardware access, scheduling, etc.) with a very specific user space. The main responsibility of that user space is to provide the Android runtime (ART) for all the Java-based components in the system, primarily the core managers and services of the Android framework itself, but also the user-installed apps. The Android framework does not solely consist of Java components, many core features are provided by native implementations that are written in C and C++. These are either accessed with the use of the Java Native Interface (JNI) or via binder IPC. While a native implementation provides various benefits with respect to performance and size of the application, it is also the implementation type of the missing link in this description between the Android framework and the Linux kernel. The framework and, by association, the apps do not communicate directly with the hardware, a so called Hardware Abstraction Layer (HAL) is used to define the hardware interfaces and strictly separate the framework from the operating system.

The Linux Kernel as the Central Component

The Linux kernel builds the foundation of a typical Android device. Starting with a fork of the mainline kernel various mechanisms were implemented to provide the required features the Android frameworks relies on. These features include the core IPC/RPC mechanism called binder [2], a low-memory killer and the ION memory allocator. With time, the Google engineers realized more and more how crucial the work of the mainline Linux kernel community was for the success of the project and thus the development was aligned to the mainline work and the majority of additional Android features were re-implemented to be merged back into the mainline. Nowadays almost all features are part of the mainline Linux kernel with the exception of some very Android-specific components such as the interactive cpufreq governor [3] or paranoid networking [4].

Modern Android systems are based on AOSP common kernels [5], which are downstream of Long Term Supported (LTS) mainline Linux kernels. As mentioned above, these kernels include the Androidisms required by the Android framework. They get regular merges from LTS into separate branches (android-x.y) and are validated with automated test frameworks. The OEM/ODM kernels used in devices are required to merge the LTS changes from the originating android-x.y branch regularly.

While the Linux kernel requirements were not exactly version-specific in legacy systems, modern Android systems not only allow modular kernels but may also define specific versions and configurations of kernels to be used. That is particularly important for framework updates and the accompanying vendor interface tests. Consequently, the Android framework can be separated from the platform and vendor components can be developed and deployed independently.

Binder IPC Framework

The binder framework is the core IPC mechanism used in Android systems that extends the traditional IPC facilities usually encountered in Linux systems. It offers more than just simple IPC messaging, most prominently it allows remote procedure calling (RPC) on methods for remote objects as if they are local object methods.

Fundamentally, the binder communicates in a client-server model, with a thread pool waiting for requests on the server side and connecting clients that synchronously connect with it using proxy objects provided by the binder framework. These proxies allow the clients to be agnostic about the

implementation details of the communication and are responsible for the actual data transfer. A central component in the binder framework is a so-called context manager, the special binder node that is assigned the ID 0. Its function is comparable to a registry, a known location to get information about other communication participants. Thereby, communication partners do not need to know details of each other with respect to their location in the system and instead receive an interaction handle for interaction.

There are a lot more details to the binder including binder tokens that uniquely identify communication partners in the system, reference counting to allow system resources to be freed automatically or parcels, the implementation of object serialization, but a simplified summary of the binder is as follows: the binder allows processes to communicate, and data transfer with and to each other and every Android-related communication makes use of the binder, even though this is not always apparent. The key is that none of the details are of concern to the components in an Android system, as the code for performing the IPC is generated during the build time of the system based on interface description languages (AIDL, HIDL). Communication partners simply interact with managers to retrieve handles to perform actions upon the system or register themselves (basically in form of callbacks) with it.

Hardware Abstraction Layer (HAL)

Even though Android uses the Linux kernel to interact with hardware and the kernel in turn provides an interface to do so, lack of standardization and stability of driver interfaces require the definition of standard interfaces for vendors to implement against. These interfaces enable the Android framework to be agnostic about the lower level implementations of the hardware interaction. Generally, a HAL implementation can be seen as a module which is typically a shared library that is loaded on demand by the system.

As it is very likely that legacy Android systems will be encountered in the industrial environment, the following chapter briefly describes the former mechanism for HAL implementations followed by the current architecture used to describe hardware interfaces.

Legacy Android HALs

As described in the previous section, a HAL consists of an interface description and requires a component that implements that interface to be usable by others. Typically such a HAL is implemented with the use of code generators. Before Android 8 the situation was somewhat different. With the origin of Android in the C programming language the interface descriptions were based on mere C header files defining structures for the HAL modules and devices to be implemented by the vendor. As such, these interfaces were very static to begin with. Figure 2 shows an example of the legacy light HAL interface, which consists of an embedded structure used to implement inheritance and a method signature.

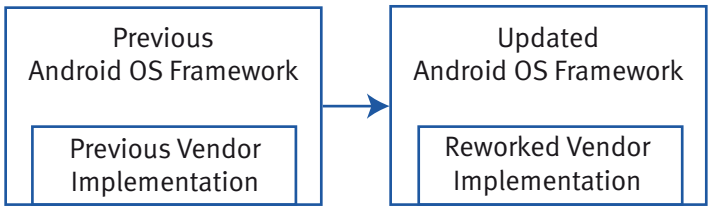
```
struct light_device_t {
        struct hw_device_t common;

int (*set_light)(struct light_device_t* dev,
        struct light_state_t const* state);
};
```

Figure 2: Legacy structure of the Light HAL

When updating older Android systems, changes in the HAL interfaces directly forced the vendors to verify for every single HAL implementation that all the method signatures and structures are still valid and being used correctly. Otherwise, refactoring or reimplementation was necessary, starting with simple new functions that are used by the framework, through new structure attributes to deprecation of components in the HAL itself. That led to very low motivation of vendors to perform Android version updates due to the high expected costs with minimal benefit for the current generation of devices. Additionally, it was eminently difficult for a framework component to identify the actual interface version that was implemented. The framework services call into the available HAL components by loading a shared library of a specific name and then they call its functions, uncertain whether that library implements the HAL of one or the other Android version.

Project Treble – The Necessary Re-architecture

Android Updates before Treble



Android Updates after Treble

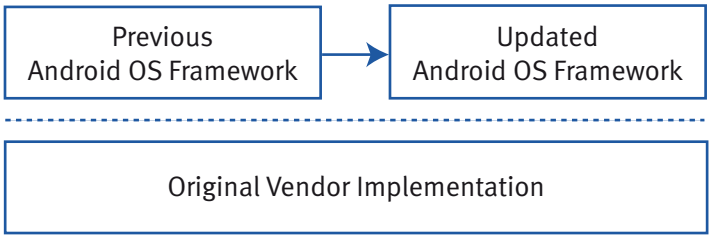


Figure 3: Android updates before and after Treble

With the release of Android 8 Oreo a re-architecture with the name Project Treble was introduced. The main goal is to separate the Android framework from the hardware-specific parts of the system, most prominently to allow independent updating of the Android framework without the need for vendors to rework their implementation. Figure 3 illustrates the situation before and after Treble. Before Treble vendor implementations were largely influenced by framework updates whereas with the new architecture the original implementation can be used beyond an update.

A modern HAL implementation can be more or less agnostic about the rest of the system. As mentioned in section „Hardware Abstraction Layer (HAL)“ the interface is defined in a specific location and the generator „hidl-gen“ is used during build-time of the project to generate the boilerplate code that handles the binder communication and data transfer. With the generated proxies, the Android framework can make use of the interface functions by acquiring the implemented service from the aforementioned context manager. The stubs (which are the interface functions) are implemented by the HAL itself and the binder is responsible for performing the actual RPC. As an example, the thermal HAL in Android 10 provides HAL interfaces in versions 1.0, 1.1 and 2.0. Version 1.1 extends the base version of the interface with a mechanism to register a callback when thermal values change. Version 2.0 reuses large amounts of the 1.0 API, but restructures the basic structures used in the interface, which is the root cause for the interface incompatibility. The Android framework component that uses this interface is the ThermalManagerService. During its initialization, this service checks explicitly which HAL implementation it can connect to from newest to oldest. So in this case, the

```
package android.hardware.light@2.0;

interface ILight {
    setLight(Type type, LightState state)
                        generates (Status status);

    getSupportedTypes() generates (vec<Type> types);
};
```

Figure 4: HIDL Interface of the Light HAL

aforementioned callback introduced with version 1.1 might not be available in the system this framework is executed on and, depending on the version, the data structures might differ. It is now the responsibility of the framework service to provide a valid differentiation between all supported HAL versions.

For HIDL-based IPC, first and foremost the interface needs to be described with the interface description itself. While conceptually similar to the legacy HAL description as shown in figure 2, figure 4 shows how such an interface is described in HIDL, in particular the aforementioned versioning information. Based on the interface itself a client-server-like architecture is realized. The server is the component in the system that implements the HAL itself and receives the calls noted in the interface. The clients use that interface and make calls on the provided methods.

Figure 5 illustrates the interaction of a HAL client and server based on HIDL. This interaction is possible through the generated code that provides proxies and stubs and the server does not need to care about any of the specifics behind the communication itself. It simply provides the callbacks defined in the interface and registers itself as the provider for that specific HAL interface with the service manager. From there on any component in the system can acquire a handle to the registered implementation through the service manager and call its functions as though they were its own.

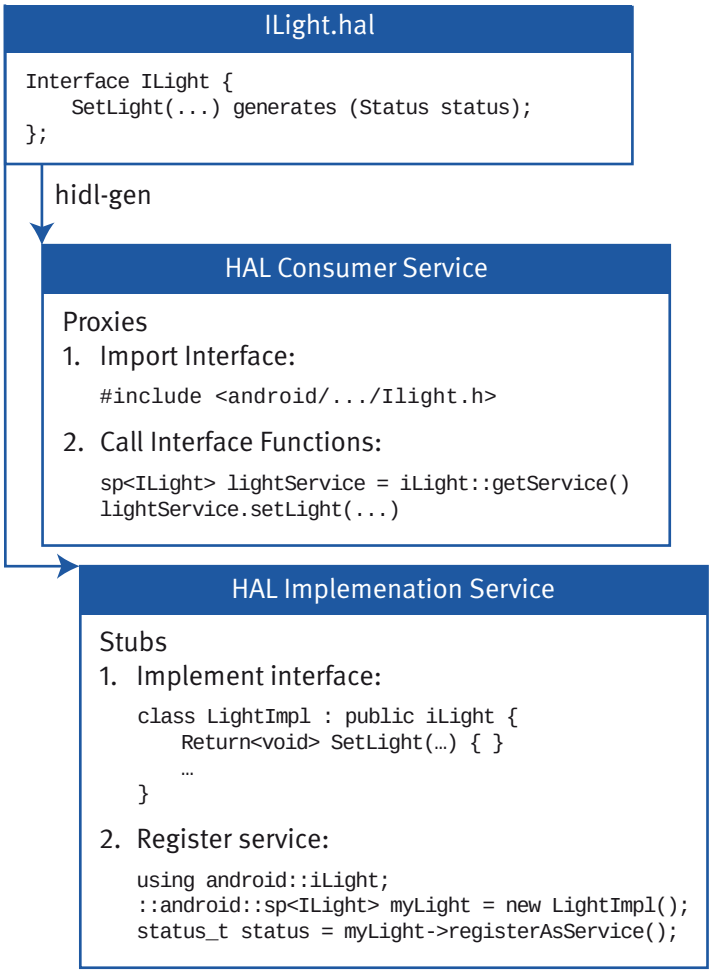


Figure 5: HIDL-based IPC

Android Security Features

Android features a multitude of different security features which are mostly centred around established mechanisms provided by modern Linux systems and profound concepts of embedded industrial systems. Many of these features, notably those related to the hardware the Android system is executed on, are not necessarily required, as different hardware configurations may not even allow the use of, for example, a trusted execution environment (TEE).

Based on process isolation in Linux systems, one of the core mechanisms used is the assignment of separate user and group IDs

for every process in the system. That alone allows the system to be designed very explicitly regarding the components that interact with each other as their system resources are not accessible by one another without consent. Moreover, the central entity controlling every process interaction on the lowest level is the Linux kernel itself, as the binder has to be used for that interaction.

The Android framework provides central software-based security features such as an app permission model, app signing, and authentication based on a cryptographic key store that can also be accessed through biometric factors. These features, while very important for the security and integrity of an Android system, are not the focus of this paper as it will focus on the features that are relevant and have to be considered when porting and updating Android devices.

In addition to the software-based security mechanisms of the Android framework, several hardware features are also used to provide strong device security. From the earliest moments of a device startup, the secure boot process of Android is executed. Google calls it Android Verified Boot (AVB) and it is based on the dm-verity feature of the Linux kernel. The main idea is to cryptographically sign all boot stages (bootloader, kernel, root file system, etc) and ensure the overall integrity by letting every stage verify the next. Finally, dm-verity is also used to ensure the system integrity during runtime with the use of a hash tree. AVB also provides mechanisms to communicate the verification state of the system to the user and rollback protection.

The next major security component is the TEE implementation used by Android called Trusty TEE. It is a secure OS running on the same processor as the Android OS but is completely isolated from it thanks to the ARM TrustZone Technology [6]. The TEE is responsible for the most critical operations such as password verification, secure key storage or digital rights management. It is not mandatory to make use of this exact implementation, the AOSP contains reference implementations for the relevant HAL implementations (gatekeeper, keymaster) that can work with Trusty out of the box. It is completely feasible to use a single trusted application, for example for signature calculation.

Last but not least, Android uses Security-Enhanced Linux (SELinux) to enforce mandatory access control (MAC) in addition to the aforementioned discretionary access control (DAC) based on users and groups. SELinux is implemented as part of the Linux Security Modules (LSM) framework and allows the Linux kernel to enforce rules directly on kernel objects representing resources and processes in the system. Actions on these objects can be evaluated and prevented by a central authority. The basis for the decisions is policies separated into different categories, for example for files or apps. An SELinux-based system can be configured in different styles, either as an open system or in the case of Android a denial-by-default system. In the latter, the policies specify every single action that is allowed and these rules need to cover every single component in the system. A new application added to the system would not be allowed to access any relevant components even if it would be allowed using it‘s user and group permissions until a specific policy is installed that whitelists those accesses.

Porting Android to Industrial Devices

Porting an Android system to a new device requires several steps typically performed by different entities. Generally every porting process starts with a source tree based on the AOSP. That tree provides the overall framework of the system internals and especially the expectations of, and interfaces to, the operating system. Last but not least, the exact Android version is defined by the revision of the source tree.

The foundation of an Android porting is laid by SOC vendors. Their aim is to provide the basic building blocks for Android systems based on their specific hardware. Similar to regular embedded Linux systems, first and foremost, the hardware needs to be supported by the Linux kernel. That leads to the first decision a vendor has to make: deciding on the kernel tree and version. As more modern Android variants require specific kernel versions that decision is somewhat restricted. Either way, the SOC components and drivers for the processor, memory, graphics, audio and other peripherals need to be developed and integrated.

With a set kernel version and the required drivers and configurations available in the Linux kernel, the next step, still for the SOC vendor, is to provide HAL implementations, which again require the decision on an Android version to be used in the development. As outlined in the section „Hardware Abstraction Layer (HAL)“, depending on the Android version, quite different implementations are required for a HAL. Typically nowadays the vendor will opt for the most recent Android version and implement the required components based on HIDL.

A multitude of benefits were introduced with Project Treble for the SOC vendor. Thanks to interface versioning and the service-based nature of HALs, the implementation can be completely separated from the framework and can be developed based on different Android versions. At this point the vendor decides which versions of a HAL interface will be supported. Typically the latest version will be chosen. Ultimately, the goal of the SOC vendor is to provide a bare Android system with essential hardware support.

The OEM vendor then typically creates a product design based on an SOC appropriate for his use case. This selection usually also depends on the availability of an Android porting in the required target version as the OEM does not always have the capability to develop all HALs on its own. However, the OEM still has to develop and integrate the required software for the additional peripherals of the final product hardware, which essentially differentiates the development of consumer products from industrial products. While most of the Android-typical peripherals are used, many domain-specific ones need to be integrated as well. Similar to the SOC vendor, the OEM therefore needs to develop and integrate the necessary HAL implementations. Moreover, the customizations specific to the product line are performed. Typical examples are themes, boot screens, specific vendor apps or the disabling of default components provided by the framework.

Security and Feature Updates of Android Systems

There are two kinds of Android system update: security updates and feature updates. Additionally the latter can be divided into vendor updates that contain vendor IP updates such as new firmware for a microcontroller and Android updates that include updates based on the AOSP tree resulting in version updates. As vendor updates are highly individual, they are outside the scope of this work.

Security updates are usually based on the Android security bulletins published by Google. These provide fixes for potential issues in the framework components as well as for some major chip manufacturers such as Qualcomm. However, the device manufacturers might also publish security-related fixes specific to their products directly through separate channels. Such a fix is mostly described by a specific commit in the AOSP tree as applied in a later Android version. Ultimately, the security fixes are usually broken down into monthly sets of fixes to be integrated into Android product trees. When a security bulletin for a month, e.g. April 2020, is integrated, the security patch level of the device can be updated to the new level, e.g. “2020-04-01”. The security patch level is separated

into two categories, identified by the day of the level set to 01 or 05, e.g. “2020-04-05”. The difference is that 01-levels may only address the issues in this bulletin alone, 05-levels ensure that all previous issues are covered as well.

Updating the Android version can be a very extensive operation. The customizations performed to the old version not only need to be integrated into the new tree but also re-evaluated with respect to their requirements, implementation and applicability in general. This also extends to the OS level. Hardware interfaces may have changed; in newer Android versions new interface versions may have been added as well as completely new interfaces. However, when working with post-Android 8 systems and proper HIDL-based HAL implementations, the hard requirements for re-implementations of HALs are quite limited. The complexity of HAL level updates varies depending on the actual Android version difference between the old and the new system. Updates of pre-Treble versions often pretty much resembled a complete implementation as lots of details might have changed in an interface.

The introduction of versioned HAL interfaces with project Treble was meant to address one of the central problems that Android devices were previously subject to: the lack of feature updates and even security updates. Many vendors did not ship updates to newer major Android versions, or even patch releases, to avoid the cost of updating and testing their custom code with these newer releases. This results in the fact that today hundreds of millions of devices are still in use that are affected by known security issues for which patches are already available. This situation is widely known and harmed the reputation of the Android project itself. The lack of feature updates also led to consumer devices becoming obsolete at shorter intervals, adding another factor to customer dissatisfaction.

With Android 10, the separation introduced in Project Treble is enhanced to allow even more fine-grained component decoupling. Various components are introduced and refactored into so-called “modules”, which can now be updated independently and also based on a store, e. g. the Google Play Store, instead of a fully fledged OTA update package. To allow such a separation the Android Pony EXpress (APEX) container format was introduced. These modules are based on manifest files that describe the component in version and additional information and can be used in synergy with existing tools in the Android ecosystem. Essentially, a component based on APEX is used by mounting the most recent version available to a specific location. In Android 10 not all framework components are refactored into modules, table 1 shows a list of available modules.

Android Runtime	Runtime module for native and managed Android services, contains core libraries and helpers
Captive Portal Login	Separate system app to manage logins on captive portals
Conscrypt	Offers public cryptographic functions such as key generators, ciphers and message-digests
DNS Resolver	Centralizes DNS lookups into one module, protection of DNS interception
DocumentsUI	Provides permission-based access to documents
ExtServices	Framework components that are permanently running
Media Codecs	OMX [7] based updateable codecs
Media Extractors and Media2 APIs	Multimedia framework

Module Metadata	Provides module information to the system
Network Stack Permission Configuration	Separate permission to allow network modules to perform network-related tasks
Network Components	The whole network stack (including DHCP and IP)
PermissionController	Runtime permission handling and tracking, management of roles
Time Zone Data	Daylight Saving Time and time zones

Table 1: APEX-based system modules in Android 10

Despite the challenges when updating the security patch level, generally integrating component updates or updating the complete Android system as a whole, all these tasks eventually require a deployment method, which the product vendor needs to implement. The AOSP does not have any strict requirements on how to perform updates but, except for the very recent mechanism of APEX containers, all updates are fundamentally similar as the goal is to update the system (and vendor) partition of the device. How exactly that goal is achieved does not matter to the system itself. However, Google offers a somewhat recommended update strategy based on the build system and reference implementations in the AOSP. Accordingly, there are tools available to create update bundles, either file-based (Android 4.4 and earlier) or block-based (Android 5+) and either full system updates or incremental updates.

Besides this, the tools for executing an update are available in the AOSP. One of the flexible design decisions Google made was that an update bundle brings its own instructions for executing the update with it. Thus the actual updater is more or less an interpreter that reads the update instructions contained in the update bundle and executes them. How and when they are executed differs from Android version to version. In Android version 7 and earlier, a recovery mode was used, a separate small system partition whose sole purpose is to execute an updater for update deployment. When executing an update, the system reboots into the recovery mode and executes the update. The system may or may not interact with the user during this stage and ultimately the device boots into the new updated system.

In modern Android systems the update process is called seamless update. In this process, a so-called update engine is executed at runtime in the background, deploying the update. Consequently, as an updater is unable to update the system it is executed from, the concept of slots was introduced. These are sets of partitions, one active and at least one inactive partition set. The inactive slot is updated by the update engine and upon completion the device reboots into the new system. That mechanism greatly increased not only the user experience as the most extensive operations are performed in the background but also fail safety as with a working active slot a failed update still results in a booting device using a fallback system.

Last but not least, the update packages need to be transferred to the device by some means. Again, no strict requirements are enforced by the AOSP. Updates are typically initiated based on the location of the update bundle within the file system. In modern Android systems streaming updates are also supported by the update engine, based on HTTPS URLs.

Case Study: Porting of a Secure Public Transport Ticketing System

This example outlines the security-related challenges and considerations when creating industrial products based on Android and shows how decisions reached very early in the product cycle have a significant impact on the system flexibility and quality based on the amount of time and money one is willing to invest. It is worth noting that these decisions were only possible due to the separation introduced by Project Treble as described in the previous chapters.

A product line for a ticketing system was developed based on Android 8 with extensive support from emlix. The system was planned to be maintained for at least five years and should be kept as up-to-date as possible. This requirement demanded the provision of a maintenance concept and a general architecture that could be updated in the future as cost-efficiently as possible.

The initial concepts for porting the system are fundamentally identical to the tasks described in the section „Porting Android to Industrial Devices“. To provide a successful port to the target hardware all the required components that need to be integrated have to be identified. That includes vendor-specific hardware modules, services or apps. The most important decision at this point is whether or not the framework itself will be kept unchanged as much as possible. The main reason for this is the consideration of the capability of the system to be updated in the future. Every change to the framework needs to be integrated in newer versions as well. The customer decided in this case to spend more time on a clean separation, although the inclusion of the component into the framework would have resulted in a quicker and thus cheaper solution. The complexity arose in a small but significant re-architecture of a core service.

With the system foundation in place, further security-related decisions need to be made that are highly dependent on the targeted environment of the product. One of the first issues is the identification of the user accessing the device. With consumer smartphones the users have full access to the device and are able to install apps arbitrarily, not only from trusted but also untrusted sources - which is the fundamental design of the Android operating system. In industrial domains, the use case is often quite different. In this particular case only two existing apps were to be preinstalled and exclusively usable. This defined environment allowed enormous hardening as shown below.

As described in the section „Android Security Features“ there are a multitude of security features available in the Android ecosystem. Many of these are not required to be used, depending on the Android version being utilized. Unsurprisingly, every feature that is implemented increases the system complexity and the cost of product development. In the case of the ticketing system the customer decided that a TEE or disk encryption were not necessary, but SELinux and verified boot should be implemented as the system would be used in public spaces and might be accessible by untrusted entities. Additionally, there was a vendor component that was accessible from a remote system and thus favoured stronger internal hardening.

Implementing verified boot requires a suitable partitioning scheme and an update concept to be defined, as this also defines what components need to be secured and ultimately how they are updated because their stored signatures need to be updated as well. For verified boot the AVB implementation was used. The system had cryptographic public keys burned into fuses that allowed the bootloader to perform high assurance boot (HAB), which is a feature of NXP processors. With a verified bootloader the root of trust can be ensured when booting the device and consequently the verification information can be propagated into the Android system. Besides this, the device used a

fairly simple partitioning scheme resulting in a lower complexity for the chain of trust and management of verification metadata.

The greatest effort was necessary when hardening the system with SELinux. Google defines an enormous set of rules to account for all their components in the system and enable every interaction of their needs to be explicitly allowed. In Android 8 these rules are enforced by default and extend across all system components. That means that for every vendor application every interaction with the system needs to be expressed and allowed in SELinux rules explicitly. The issue that may occur, as was the case for one of the preinstalled apps in the ticketing system, is that seemingly unimportant things, such as on which partition an app is installed, can quickly create conflicts with basic rules defined by Google. There are so-called neverallow rules, for example that apps may not access specific socket types or send signals to anything other than apps that create the foundation upon which all vendor rules are based. Again, it is easily possible during development to simply modify the predefined rule set and, for example, remove the cardinal rule preventing components installed on the vendor partition from accessing files in the data partition. That, however, prevents the system from being compliant with Google‘s policies and being certified. Also, an additional integration step in any later system update is created as that rule comes directly from the Android framework itself. It could also result in further security issues arising, as all other rules are developed under the assumption that these core rules are in place.

Instead of being able simply to change a small logic within one of the preinstalled system apps, a fairly complex rule set needed to be created to circumvent these restrictions. Moreover, to create a good rule set for the components that are used in a system, all of its behaviour needs to be known and tested during development. That can be an issue when IP restrictions exist or the full spectrum of access requires additional infrastructure that is not easily accessible.

Despite this effort, SELinux also provided very strong handles to harden the system. With the aforementioned very limited set of two applications executed in the system, every other component could be restricted. Starting with banning the execution of any untrusted app at all, many of the typical allowance rules could be overridden, such as communicating with the system server.

As introduced in this section, the development of the ticketing system required multiple decisions that mostly increased the development cost and time in exchange for the prospect of higher flexibility and quality of the system. The customer was convinced that his investment would pay off in the first update cycle of the platform. In early 2020, the customer ordered an evaluation for a system update to Android 10 and the result confirmed this expectation to a great extent. Due to the carefully implemented separation of the vendor-specific components, the expected effort was reduced to half compared with a typical system update of similar magnitude.

Case Study: Updating Legacy Android Systems and its Challenges

To maintain and enhance the security of inventory products in the field, the systems need to be able to be updated and, as introduced in the chapter „Security and Feature Updates of Android Systems“, there are two update variants: updating the security based on the current version deployed or updating the version of the Android framework. Both options can be illustrated with an example case to which emlix was assigned in early 2020.

This example is based on an industrial automation control product a customer used for a large colour picker system. Besides the typical benefits, the product stood out due to its ability to integrate

new apps for recipes and compositions. It was developed with Android 6.0 and consisted of different industrial interfaces and protocols such as CAN, Ethernet or Beckhoff Fieldbus as well as the openness for the user to install various apps during the lifetime. The system was highly dynamic and a solid product for the customer to maintain for many more years. His customers, however, were apparently more and more dissatisfied with the age of the Android version and based on that pressure, a two-part task was defined. First, the system should be updated to the latest security patch level available at that time and, second, an evaluation of an Android version update to version 8 should be performed and the estimated effort identified.

In the chapter „Security and Feature Updates of Android Systems“ the basic tasks of updating a system to a recent security patch level were introduced. That process, however, not only includes the implementation of the security bulletins but also updating the components in the system. Starting with the Linux kernel the most recent patch level of the current kernel is included. In this case the version in the system was 3.10.49 and the latest version supported was 3.10.108. Such an inclusion is most likely without problems, as no interface or feature of the kernel changes from patch level to patch level.

After the kernel, the vendor components need to be evaluated such as HAL implementations from the SOC vendor or those created by the customer himself. In this case the SOC vendor stopped supporting the platform during the Android 6 release and only a few components were updated after the initial release the customer used for his product and consequently the implementation effort was negligible.

Every patch contained in a monthly bulletin needs to be evaluated and integrated. The evaluation requires a fairly good understanding of the use of the system to be able to rate the actual impact of a patch series. With about two years of security levels, separated into 24 bulletins, each consisting of 20 to 60 issues to be fixed, there is a lot of code to evaluate and understand. Furthermore, with increasing time between the release of the Android version in use and the date of the bulletin to be implemented the required effort increases, in some cases significantly. The problem is the differing code base the patches are applied to. Bulletins created for a component that has been refactored in the meantime require the developer to dig into the current version of the component and try to understand whether the issue in question is still present after the refactoring. Even if this is the case, the patch itself then needs to be applied, which may require significant adjustments to the source. In this case, implementing security bulletins starting from Android 7 proved to be fairly difficult, but with increased deviation between the bulletins and the code base the number of patches that were not applicable increased as well. Ultimately, the required effort for implementing security bulletins is extremely difficult to estimate, not least because the actual quantity of patches per bulletin varies a lot, but also because the number of applicable patches depends on the actual hardware components used. Aside from that, the security patch level does not contain any information on platforms not directly supported by Google.

The evaluation of the system update to Android 8 started with taking inventory of the system components. Basically, the full stack was influenced by changes made by the vendor, starting with a custom kernel driver that interacted with a microcontroller and provided a great deal of functionality to the system, that functionality was integrated into the system with multiple services and managers that in part made use of JNI to interact with the kernel interface directly and in part used other system managers to accumulate required information. A custom app for system control was, of course,

integrated as well. Even the updated kernel version 3.10.108 was very likely to create some issues during the update as Android 8 requires at least kernel version 3.18.x.

To identify the required adjustments we started with the foundation, the Linux kernel. Despite the aforementioned version requirement, it was originally extended by two patch sets including drivers for the WiFi and Ethernet chips used and not supported by the tree directly. Additionally, the WiFi chip and driver were discontinued so that a more recent driver version was not available and some sort of driver porting was required as a kernel version update was inevitable. Next, the implemented HALs needed to be evaluated. Various HAL implementations were included by the SOC vendor supporting the hardware components used. For that SOC however, Android support discontinued before the release of Android 7, so no updated HAL implementations for the targeted version were available. For an inertial module unit (IMU), a HAL was integrated into the source base, which, in contrast, was available in a recent version. Based on the first two layers alone, a relatively high effort would be necessary to update the complete system. It was not even clear whether other updated HAL implementations required a kernel update as they may have relied on kernel interfaces introduced in later driver versions. It was certain at this point that porting or reimplementation of one driver and multiple HAL interfaces was necessary.

As mentioned earlier, the system contained a service and a manager to support functionality provided by the microcontroller. These were integrated directly into the Android framework and as it turned out, some framework-internal functionality was required for proper operation of the manager. This meant that for the successful inclusion of the manager the exact same information would have been made available in the new system. As it turned out, for example, the PowerManager did extend the API in later versions to support a forced shut down, but the implementation and interface differed so that significant changes would have been required in that area, too. Last but not least, the public API was modified as well, not only with the inclusion of a custom manager and service but also some private API information was exported. All in all these modifications to the framework would have required not only major modifications to the new framework (e. g. because of different internal APIs) but also to the vendor components.

As a consequence of the evaluation, the customer decided that the cost for releasing the product line with a more recent Android version would not be covered by the expected revenue from the product. This brief summary illustrates how the openness of the Android system allows great flexibility in the implementation of vendor systems but also shows the limits when deviating from the system implementation as Google intended it.

In retrospect, there are not many things that could have been done differently during the initial implementation to ease the process of updating the framework to a more recent version, due to the limitations of Android itself at the time. However, even in the case of the fairly large version difference at hand, some implementation details do benefit a system update. There were no practical solutions at the time for ensuring a development of the platform that would be easily replaceable because of the nature of legacy HALs. However, there were already mechanisms available to include vendor components and make them available to vendor apps without modifying the framework itself. That still would have required adjustments during a framework update, but the expected effort would have been far more manageable.

Both update strategies, the security update implementation as well as the full Android porting, would follow the same deployment path as they are both implemented as full partition updates not based on the reference implementations available in the AOSP. The customer uses a company and

product-wide update strategy that is implemented as a native binary, which securely acquires the update bundle from an update server and performs the update. The platform already featured a multiple partition set scheme where the updater installs SquashFS images in inactive partitions. As the source for creating and extracting update bundles is available in the AOSP, the code base could easily be modified to produce update bundles compatible with the well-established update system of the company. It was clear to the customer that continuing to use that system would require more effort for the initial system integration or later adjustments when updating the AOSP repository but in the long run using the same update mechanism across all product lines would pay off.

Even though the legacy Android system had sufficient performance and was stable and adequately secured, this example is typical of the challenges older Android systems in the field face during their lifetime. Bringing legacy systems up-to-date can be a daunting and expensive task but when considered from the beginning of the development, the amount of self-inflicted impediments can be reduced significantly. Even though the version update for the old system was cancelled, the customer was still convinced of the opportunities Android systems offer and announced the next product of that line, which is to be developed based on the most recent Android version – with all the aforementioned longevity benefits.

Increased Attractiveness to the Industrial Market

The two case studies in this paper are fairly representative of the situation for Android-based products in the industrial market. The majority of Android systems in the field are legacy systems which are inherently difficult to maintain, depending on their deviation from the targeted development flow. The architecture was so intertwined between the framework and the platform that all components that cross that boundary in one way or another need to be at least examined and more likely to be refactored during a system update.

With the introduction of Project Treble, groundbreaking facilities were introduced to create long-lasting systems that are more easily kept up-to-date and provide the possibility to include framework updates, and consequently complete system updates, selectively and incrementally with less effort. This improvement is even more relevant in the industrial domain as the cost per unit for updating a product can be far higher compared to the consumer market due to the typically small quantity of devices sold and the fact that industrial devices typically contain even more supported and integrated hardware interfaces. This means that in legacy systems the effort required for system updates is inherently larger as more interfaces need to be updated and integrated components to be reviewed and possibly refactored, while at the same time the costs can only be spread across fewer installations.

The security mechanisms provided by Android‘s AOSP are manifold and allow vendors to include them selectively into their products. While these features generally also increase the system complexity, industrial devices face a different and more confined environment. This allows a vendor to adjust the security model to his needs.

Bibliography

[1]: Google Inc., Google Mobile Services, https://www.android.com/intl/en_us/gms/

[2]: eLinux.org, Android Binder, https://elinux.org/Android_Binder

[3]: Mike Chan, cpufreq: interactive: New, interactive‘ governor, https://lwn.net/Articles/662209/

[4]: eLinux.org, Paranoid network-ing, https://elinux.org/Android_Security#Paranoid_network-ing

[5]: Google, Common Android Kernel Tree, https://android.googlesource.com/kernel/common/

[6]: Arm Limited, Arm TrustZone Technology, https://developer.arm.com/ip-products/security-ip/trustzone

[7]: The Khronos® Group Inc., OpenMAX - The Standard for Media Library Portability, https://www.khronos.org/openmax/

About the author

Martin Homuth has studied technical informatics at the TU Berlin and has been working as a system engineer for embedded Linux and Android-based systems at emlix since 2015.

About emlix

emlix provides embedded Linux and Android system solutions for the digitalization and networking of products. The company‘s core competence is the development of software for devices, plant and machinery. In addition to the design and development of functions at the device and process control levels, our services include server-side integration into cloud and ERP systems.

Contact

Should you have any further questions, please do not hesitate to contact us:

emlix GmbH
Gothaer Platz 3
37083 Goettingen
Germany

www.emlix.com
solutions@emlix.com
+49 (0)551/30664-0