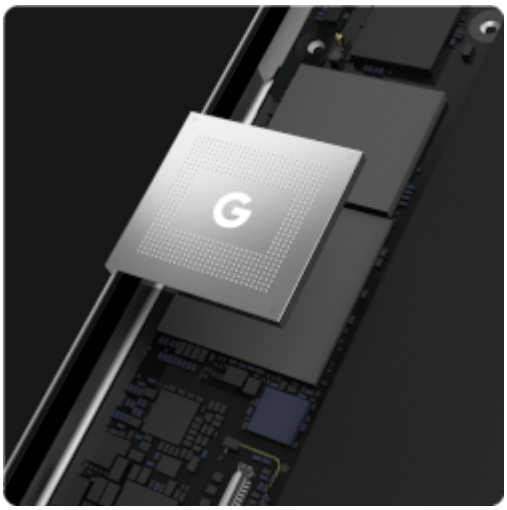
GrapheneOS - A High-level Overview

Table of Contents

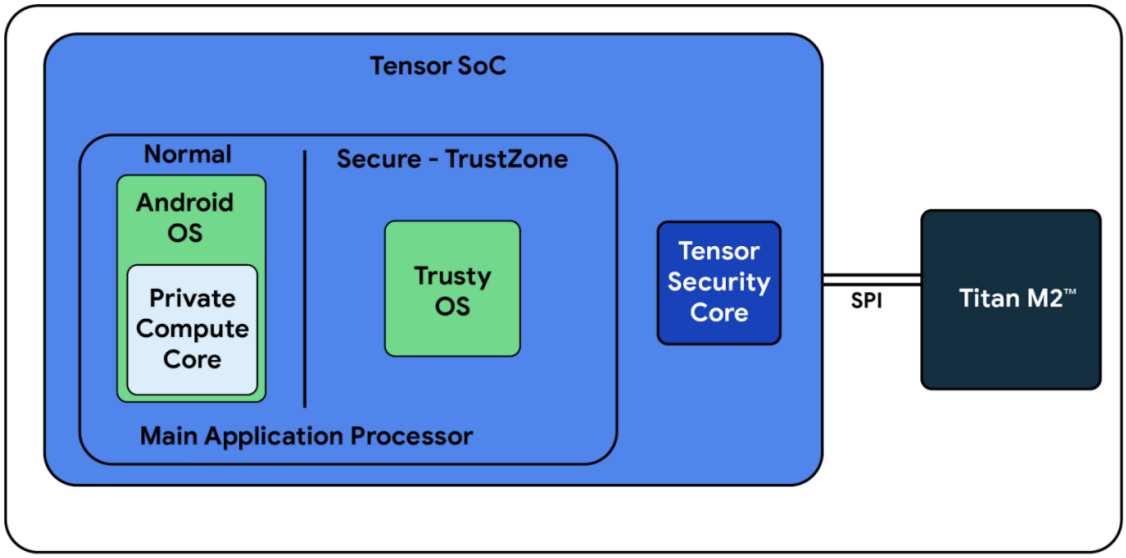
Part 1 - Hardware

The Titan M Series Chip Graphene OS Security begins at the hardware level, by utilizing The Titan M Series Chip. The Titan M2 is a dedicated security chip included in Pixel 6 and 7 series smartphones. This chip was developed in-house and is based on the RISC-V CPU architecture. It contains its own memory, RAM, and cryptographic accelerator.



This design adds another layer of protection on top of Android's default security. One such security feature is full-disk encryption, which depends on the Trusted Execution Environment (TEE). The TEE is considered the secure area of a processor, and is where Android devices store their encryption keys. This is protected by your pattern, PIN, or passcode. It isolates cryptographic keys, and is never revealed to the user or the Operation System.

As of the Pixel 3, Google separated the Trusted Execution Environment from the chipset, and implemented a separate security module in its place. The Titan M Series can almost be considered a standalone processor. Additionaly, it possesses flash memory which is used for storing sensitive data, as well as running it's own microkernel.



"The Titan M2 supports Android StrongBox, which is a safe storage space for cryptographic keys used by third-party apps. The Titan M communicates with the main application processor but does not reveal the Secret keys to the app processor. It is tamper-resistant because no known manipulation of the chip will result in successful extraction.

Rate limitation When you boot up a Pixel, the data remains encrypted until the lock screen is cleared. The Titan M series has a brute-force rate-limiting feature, which is implemented at the hardware level. It uses a rate-limit feature, that hardcodes delay periods upon unsuccessful unlock attempts.

It works like this:

- 0 to 4 failed attempts: no delay
- 5 failed attempts: 30-second delay
- 6 to 9 failed attempts: no delay
- 10 to 29 failed attempts: 30-second delay
- 30 to 139 failed attempts: 30 × 2⁽n - 30) ÷ 10⁾ where n is the number of failed attempts. This means the delay doubles after every 10 attempts. There's a 30-second delay after 30 failed attempts, 60s after 40, 120s after 50, 240s after 60, 480s after 70, 960s after 80, 1920s after 90, 3840s after 100, 7680s after 110, 15360s after 120, and 30720s after 130.
- 140 or more failed attempts: 86400-second delay (1 day)

Essentially, the delay doubles after every 10 attempts until the 139th failed attempt. After which, you get only one attempt every 24 hours.

The M2 is hardened against physical tampering too. The chip's firmware cannot be changed or updated without the device's pattern or PIN.

Insider attack resistance In order to load firmware to the Titan M, the phone must be unlocked. This makes it more difficult for the creation of custom firmware that could be used to unlock your phone. This resistance to an insider attack, means only the user's authentication can unlock it. Not until you clear the lock screen prompt will the phone's storage be decrypted.

Part Two: Resisting Persistence

Verified Boot

There are four different states of integrity

- **Green**: No issues found and the system fully boots.
- **Yellow**: The Bootloader is locked and signed with a user-defined key.
- **Orange**: Indicates that the bootloader has been unlocked, and the system's security is compromised.
- **Red**: Either no valid system was found, or the device has been corrupted

Boot Flow

The recommended boot flow for a device is as follows:

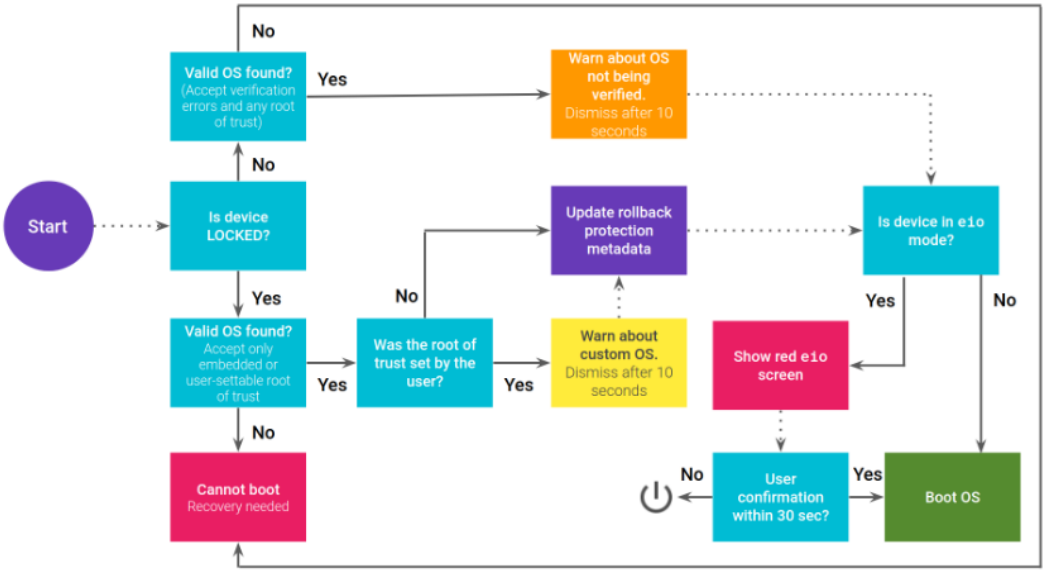


***Figure 1**. Verified boot flow*

GrapheneOS takes advantage of the yellow stage of verified boot. This is accomplished by using its own custom signing key, while keeping the bootloader locked while maintaining full verified boot. Lastly, locking and re-locking the bootloader wipes your phone's data,

Attestation GrapheneOS uses a hardware-based attestation service to validate the identity of a device, as well as the authenticity and integrity of the Operating System. This is accomplished via the auditor app.

This serves as a backup to the verified boot process. In the event an attacker were able to downgrade or tamper with the Operating System, it is the Auditor App that would detect such a compromise. This requires two devices to complete. The attestation keys are stored in the Titan M chip.

Part 3: Security Hardening of the Android Operating System

GrapheneOS Security Practices

The developers of GrapheneOS have a unique security paradigm for AOS Security. Here are a few quotes taken from their page on exploit protection features.

> *"The first line of defense is attack surface reduction. Removing unnecessary code or exposed attack surface eliminates many vulnerabilities completely."*

> *"An example we landed upstream in Android is disallowing using the kernel's profiling support by default, since it was and still is a major source of Linux kernel vulnerabilities."*

> *"The next line of defense is preventing an attacker from exploiting a vulnerability, either by making it impossible, unreliable, or at least meaningfully harder to develop."*

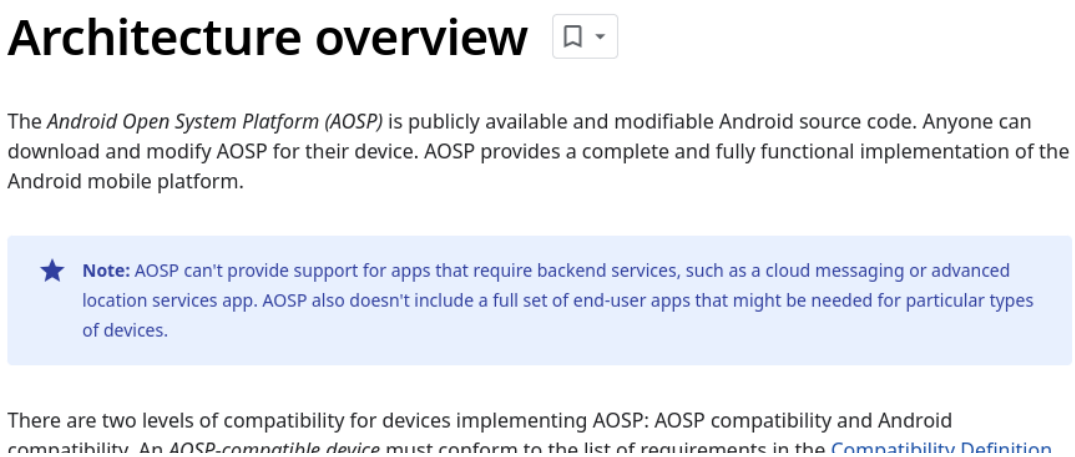
> *"In many cases, vulnerability classes can be completely wiped out while in many others they can at least be made meaningfully harder to exploit."*

> *"We offer toggles for users to choose the compromises they prefer instead of forcing it on them."*

> *"The final line of defense is containment through sandboxing at various levels: fine-grained sandboxes around a specific context like per-site browser renderers, sandboxes around a specific component like Android's media codec sandbox, and app/workspace sandboxes like the Android app sandbox used to sandbox each app which is also the basis for user/work profiles. GrapheneOS improves all of these sandboxes through fortifying the kernel and other base OS components along with improving the sandboxing"*

Defense in Depth is a at their core. Let's take a high-level overview of The Android Operating System and see what the developers at GrapheneOS do differently.

The Android Software Stack and Platform architecture



Architecture overview

The *Android Open System Platform (AOSP)* is publicly available and modifiable Android source code. Anyone can download and modify AOSP for their device. AOSP provides a complete and fully functional implementation of the Android mobile platform.

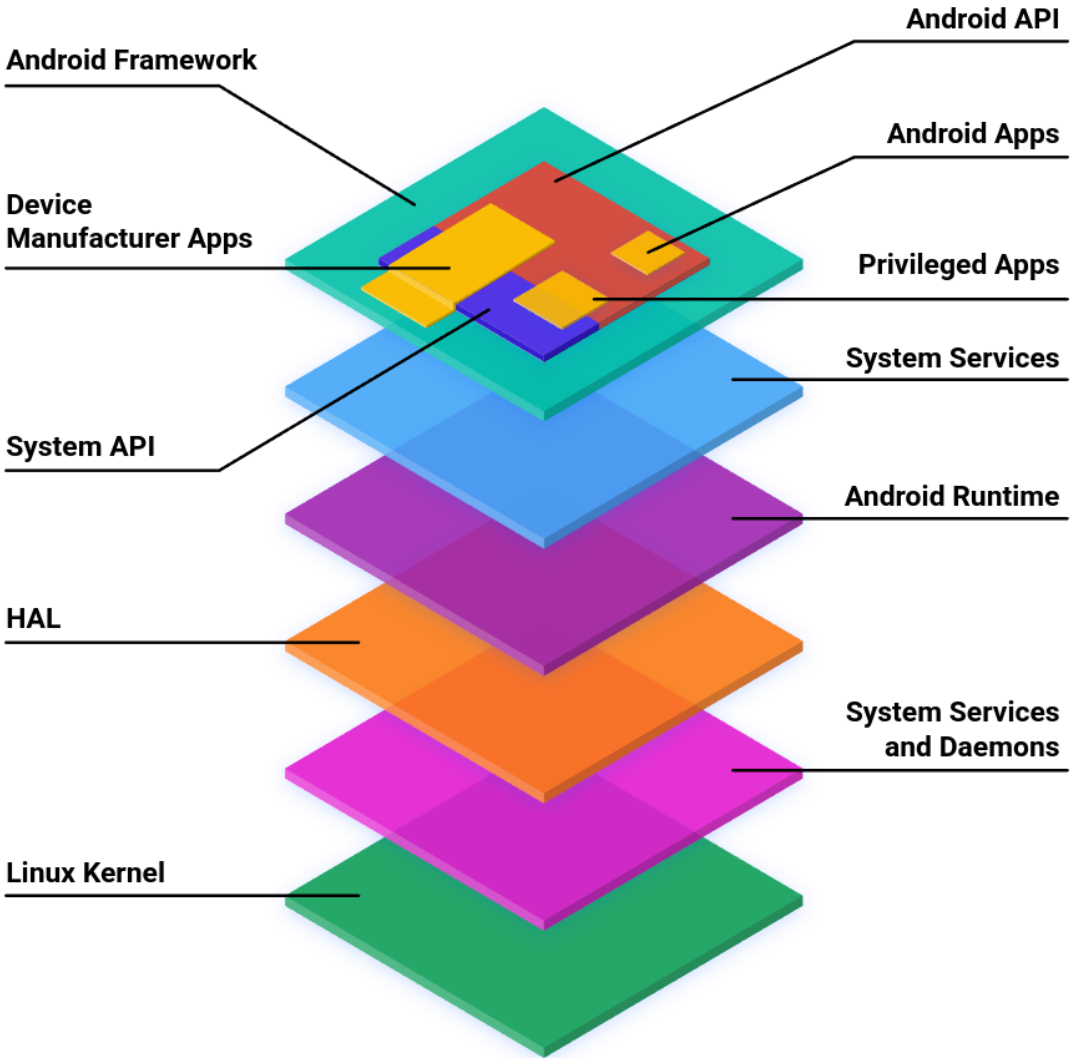
Note: AOSP can't provide support for apps that require backend services, such as a cloud messaging or advanced location services app. AOSP also doesn't include a full set of end-user apps that might be needed for particular types of devices.

There are two levels of compatibility for devices implementing AOSP: AOSP compatibility and Android compatibility. An *AOSP-compatible device* must conform to the list of requirements in the Compatibility Definition

compatibility. An *AOSP-compatible device* must conform to the list of requirements in the Compatibility Definition Document (CDD). An *Android-compatible device* must conform to the list of requirements in the CDD and Vendor Software Requirements (VSR) and tests such as those in the Vendor Test Suite (VTS) and Compatibility Test Suite (CTS). For further information on Android compatibility, refer to the Android compatibility program.

AOSP architecture

The software stack for AOSP contains the following layers:



Contents of each layer

Take note of the purple section - *Native C/C++ Libraries*, this will come up later in this review.

Application Sandbox

Android relies on a number of UNIX style protections to enforce the application sandbox. According to The Android Platform Security Model's official documentation. These are some of the areas where GrapheneOS improves upon.

- **Discretionary Access Control (DAC):** This serves as the first security control each app runs with its own user ID (UID) on UNIX-based systems. Apps can decide who can access their resources by changing permissions on those resources. Apps can also share access to their resources with other apps by passing a handle to

the resource through Inter-Process Communication (IPC). However, if an app is running with root user privileges, it can bypass these permissions, although there may still be some restrictions imposed by Mandatory Access Control (MAC) policies.

- **Mandatory Access Control (MAC):** This is the second layer of sandboxing. Mandatory Access Control (MAC) means that there are strict rules about what actions can be taken on the system. Only actions that are specifically allowed by these rules are allowed to happen. On Android devices, this is managed using SELinux which uses default denial. Meaning, anything not explicitly granted access is denied. This feature is built into the core of the operating system. SELinux is heavily used on Android to protect different parts of the system and ensure that it meets certain security standards. With SELinux, even processes that have the same user ID (UID) are kept separate, providing a more detailed level of security. This makes it more difficult for remote exploits to take advantage of system process of access user data. This is because it would have to bypass SELinux in the Kernel.
- **Android permissions** Android permissions are like special permissions that control what different apps on your phone can do. These permissions include things like accessing your location or using your camera. Each permission is assigned to a unique ID (UID), and apps can grant access to specific pieces of data they manage. The enforcement of these permissions is mostly handled by the app or service providing the data, although there are some exceptions, like internet access, which is handled differently. These permissions are set up in an app's settings file called AndroidManifest.xml and are the main way users see and control what their apps can do.

Security hardening of SELinux

What does GrapheneOS do to enhance the security of SELinux?

They split Trusted, and Untrusted app domains.

By default, the base operating system, and all the apps you install, are running in the, "untrusted app domain". But, in GrapheneOS, they are split into two separate categories: **"untrusted app"**, and **"untrusted base-app."**

The **untrusted base-app**, is the SELinux domain, used for all apps that come with the operating system. One of the changes Graphene makes to the untrusted base-app, is the removal the ability for applications to have writable and executable memory.

Android Permissions categories, and Graphene's improvements

Android permissions's five categories of consent:

1. Install time permissions
2. Runtime permissions
3. Special Access permissions
4. Privileged permissions
5. Signature permissions

Graphnene forces all apps to submit to the same sandbox policy. Privileged apps that would normally run with extensive access on Android, will be restricted on GrapheneOS. This means, that you could install Google Apps on your phone, and they would not have access to your device identifiers, or data from other apps.

Also, Graphnene introduced toggles that were not available on any other operating system. These toggles allow you to disable an apps' access to various parts of your device. This includes:

- Network permission. Disable direct and indirect access to all cellular and WiFi networks.
- Sensor permissions, enables you to zero out all values for the gyroscope, accelerator and more.

One such example is when granting network permissions to an app. The toggles let you select: grant zero permissions, only while in use, or one-time only. Though features such as these, are finding their way into other mobile operating systems, Graphene is completely opensource, and regularly audited.

What about memory unsafe languages?

Hardened Malloc

Introduction Hardened malloc is one of the main security features offered by GrapheneOS. The aims of which is preventing memory corruption vulnerabilities. One of Hardened Malloc's main features is protection against heap corruption vulnerabilities. Anothner benifit of Hardened Malloc, is efficient memory management. It is scalable by way of a configurable number of independent areas, and the internal locking is within areas that are further divided up in a per size class.

This project currently supports Bionic (Android), musl and glibc. In Android, there's custom integration, and other hardening features planned for musl in the future. The glibc support will be limited to replacing the malloc implementation. Musl is a much more robust and a cleaner base to build on, while covering the same use cases.

Harden Malloc and OS integration

On Android-based operating systems, GrapheneOS's hardened_malloc is integrated into the standard C library as the standard malloc implementation. The integration code is able to be used by other Android-based operating systems too. If desired, jemalloc can be left as a runtime configuration option only.

Core design of Harden Malloc

The hardened malloc allocator is designed to be simple and focuses on security for 64-bit systems. It uses a dedicated area of memory for storing allocation info. This is to make exploitation more difficult. Small allocations are managed within a large memory region, and large allocations are tracked using a global table.

The allocator divides memory into sections based on size, and uses a bitmap to keep track of free parts. Large allocations are managed separately and mapped and unmapped directly in memory.

This allocator is intended for security and not for helping developers find and fix memory bugs. It prioritizes minimizing overhead, and maximizing security. Vulnerability prevention is it's primary purpose.

Security features of Hardened malloc

The security features of Hardened malloc can be broken into several key points:

1. **Isolated Memory Management**: The memory allocator's critical information is kept separate from regular memory to prevent corruption.

2. **Protection Mechanisms**: This utilizes techniques such as Memory Protection Keys (MPK) and, in the future, Memory Tagging Extension (MTE) on ARMv8.5+ will be available for additional security.
3. **Detection of Invalid Operations**: This detects and prevents invalid memory operations such as freeing unallocated, or misaligned memory.
4. **Randomization**: Introduces randomness in memory layout, and allocation to make it more difficult for attackers to predict and exploit vulnerabilities.
5. **Zeroing and Canaries**: Ensures memory is cleared and uses canaries to detect overflows and underflows.
6. **Quarantine Mechanism**: This places freed memory in a quarantine to detect and prevent reuse-after-free vulnerabilities.
7. **Error Handling**: Treats errors seriously to catch any issues with memory management.
8. **Memory Tagging**: This hardend malloc feature utilizes Memory Tagging Extension, (MTE) on ARMv8.5+ to add tags to memory blocks. The result is improved security against various memory corruption vulnerabilities.

In short Hardened malloc ensures memory is handled securely by keeping it isolated, protected, and detects any suspicious activities to prevent potential attacks.

Sources:

Hardware

- https://www.androidauthority.com/titan-m2-google-3261547/
- https://safety.google/intl/en_us/pixel/

Android

- https://source.android.com/
- https://source.android.com/docs/security/overview
- https://developer.android.com/guide/platform
- https://source.android.com/docs/core/architecture
- https://source.android.com/docs/security/features
- https://source.android.com/docs/security/app-sandbox
- Android Security Model 2023 pdf
- https://source.android.com/docs/security/features/verifiedboot
- https://source.android.com/docs/security/features/verifiedboot/device-state
- https://source.android.com/docs/core/permissions/filesystem
- https://source.android.com/docs/security/features/selinux
- https://source.android.com/docs/security/features/authentication

- https://source.android.com/docs/security/features/verifiedboot/boot-flow
- https://source.android.com/docs/security/features/selinux/concepts

GrapheneOS, Documentation & Souce Code

- https://github.com/GrapheneOS
- https://github.com/GrapheneOS/hardened_malloc
- https://grapheneos.org/
- https://grapheneos.org/faq
- https://grapheneos.org/features
- https://grapheneos.org/usage#sandboxed-google-play
- https://grapheneos.org/features#exploit-mitigations
- https://attestation.app/about
- https://grapheneos.org/features#auditor
- app domain, trusted & untrusted
- https://github.com/GrapheneOS/platform_system_sepolicy/blob/14/public/app.te
- https://github.com/GrapheneOS/hardened_malloc/blob/main/h_malloc.c