Secure Boot

Teilsicherheitskonzept – CONBOTICS MalerRoboter

INHALTSVERZEICHNIS

Seite

ANLAGENVERZEICHNIS

1. Zusammenfassung

Die CONBOTICS GmbH ist ein Unternehmen im Bereich der automatisierten Oberflächenbeschichtung und entwickelt hochmoderne Robotersysteme für die Bau- und Fertigungsindustrie. Der Schwerpunkt liegt auf der Entwicklung und Integration intelligenter, autonomer Roboter, wie den MalerRoboter, die durch eine Kombination aus Sensorik, mobiler Steuerung, KI-gestützter Softwarearchitektur und sicherer Kommunikation industriellen Ansprüchen genügen.

Mit dem zunehmenden Einsatz von vernetzten, softwaregetriebenen und sicherheitsrelevanten Systemkomponenten ergeben sich für CONBOTICS erhöhte Anforderungen an die Informationssicherheit. Sowohl die Integrität und Authentizität der eingesetzten Softwaresysteme als auch der Schutz sensibler Betriebs- und Nutzerdaten stehen dabei im Fokus. Insbesondere vor dem Hintergrund zunehmender Bedrohungen durch gezielte Manipulationen, Schadsoftware und Industriespionage ist ein systematisches, auf aktuellen Normen und Standards basierendes Sicherheitskonzept von Bedeutung.

Dieses Teilkonzept betrachtet das Themenfeld SecureBoot. Das Risikoprofil sowie Details zum Threat-Modell werden im zentralen Konzept zusammengefasst.

2. Spezifikationen

Nachfolgende Spezifikationen wurden festgelegt:

Die Systemlandschaft des CONBOTICS MalerRoboters basiert auf einer modularen, vernetzten Architektur, die sowohl klassische IT-Komponenten (PC, Jetson, Tablet) als auch eingebettete Steuerungen (ESP32, Sicherheits-SPS) integriert. Nachfolgend werden alle relevanten Hard- und Softwarekomponenten beschrieben, die für den Betrieb, die Steuerung und die Sicherheit des Robotersystems von Bedeutung sind.

2.1. Zentrale Recheneinheit (PC-Plattform)

Komponente	Beschreibung
CPU	Intel Core i5-12500 (Alder Lake, 12th Gen)
Mainboard	Q670EI-IM-A (Industrie-Mainboard mit UEFI Secure Boot Unterstützung)
RAM	CIR-S5SUSC4808G (DDR4, 8 GB)
SSD	CIS-2ST380MOF512GS (512 GB, Industrie-SSD)
CAN-Modul	EGPC B201 (PCIe CAN Interface zur Kommunikation mit ESPs und Motorsteuerungen)
Betriebssystem	Ubuntu 20.04.6 LTS Server Edition
Software	ROS Noetic (Robot Operating System, Middleware für Robotikapplikationen)

2.2. Bildverarbeitung und KI (Jetson Orin Nano)

Komponente	Beschreibung
Plattform	NVIDIA Jetson Orin Nano Developer Kit
Software	Jetpack 5.1.3 (L4T 35.5.0, Ubuntu 20.04)
Einsatzgebiet	Bildverarbeitung, AI-Modelle, Pfadplanung

2.3. Kommunikationsinfrastruktur

Komponente	Beschreibung
Switch	Netgear GS308v3 (8-Port unmanaged)
Router	GL-X300B (4G-Router mit SIM- und eSIM-Funktion, VPN-fähig)
VPN-Client	Netbird Client (peer-to-peer Zero Trust VPN für Fernzugriff & Wartung)

2.4. Sensorik und Aktorik

Komponente	Beschreibung
Lidar	NANS3-CAAZ30ZA1 (2D/3D Umgebungserfassung)
Kamera	Intel Realsense D456 (RGB-D Kamera zur Tiefenerkennung und Navigation)

2.5. Sicherheitssteuerung (SICK Flexi Soft)

Modultyp	Beschreibung
EFI-Pro Modul	FX3-GEPR00000 (Kommunikationseinheit)
Hauptmodul	FX3-CPU000000
I/O-Modul	FX3-XTIO84002
Ethernet-Modul	FX0-GENT00000
Safe-Motion-Modul	FX3-MOC00000 (sicherheitsgerichtete Bewegungsfreigabe)
Steckverbinder	FX3-MPL000001 (für sichere Verbindung)

2.6. Antriebskomponenten

Komponente	Beschreibung
Arm-Motorcontroller	Nanotec CL4-E-Serie (3× CL4-E-2-2, 2× CL4-E-1-2, über CAN angebunden)
Basis-Motorcontroller	Oriental Motors BLV-R (DC-Motorcontroller für Fahrplattform)

2.7. Mikrocontroller (ESP32-Nodes)

Komponente	Beschreibung
Arm-ESP	diymore ESP32 WROOM
Base-ESP	AZDelivery ESP32 NodeMCU CP2102 mit SN65HVD230DR (CAN-Schnittstelle)
Funktion	Steuerung untergeordneter Einheiten (z. B. Motoren, Sensorik), Kommunikation mit ROS über CAN
Firmware	Eigene Entwicklung, OTA-Update-fähig, geplant mit Secure Boot & Flash Encryption

2.8. Bedieneinheit (Tablet mit App)

Komponente	Beschreibung
Gerät	Samsung Galaxy Tab A8 (SM-X205)
Betriebssystem	Android 14
App-Technologie	Capacitor JS (native App mit Webview), React, TypeScript, Vite
Funktionen	Steuerung des Roboters, Anzeige von Plänen, Parametrierung

Besonderheiten

- roslibjs für WebSocket-Verbindung zur ROS-Bridge (derzeit unverschlüsselt)
- Zugriff auf Dateisystem über @capacitor/filesystem
- Live-Updates über @capawesome/capacitor-live-update (mit Bundle-Signatur)
- Tablets müssen für Updates mit dem Internet verbunden sein
- Alternative Möglichkeit: Self-hosted Bundle-Server für mehr Kontrolle |

2.9. Externe Eingabegeräte

Komponente	Beschreibung
Gamepad	Sony PS5 DualSense Controller (Bluetooth, zur manuellen Steuerung oder Kalibrierung)

3. Secure Boot - Schutzanforderungen

Der Secure Boot des Motherboards basiert auf UEFI mit aktivierter Secure-Boot-Funktion.

Hauptrechner (Intel i5 mit Q670EI-IM-A)

- Der Secure Boot des Motherboards basiert auf UEFI mit aktivierter Secure-Boot-Funktion.
- Nur signierte Bootloader (GRUB) und Kernel-Images dürfen geladen werden.
- Die Signaturprüfung erfolgt über den OEM-Root-Key, der im UEFI gespeichert ist.
- Ubuntu 20.04 Server nutzt signierte Kernel- und Initrd-Dateien (Canonical-Authentifizierung).
- Modifikationen am Bootloader oder Kernel invalidieren die Signatur und blockieren den Start.
- Die Systemintegrität wird zusätzlich durch dm-verity (Device Mapper Verity) gestärkt, welches root-Dateisysteme auf Integrität prüft.
-

Jetson Orin Nano (Ubuntu 20.04, Jetpack 5.1.3)

- NVIDIA unterstützt Secure Boot auf Jetson durch den NVIDIA Secure Bootloader (nvtboot), welcher standardmäßig in der Jetpack-Distribution enthalten ist.
- Aktivierung des Secure Boot erfordert das Flashen von PKC- und SBK-Schlüsseln via odmfuse.sh, um den Bootloader zu versiegeln.
- Der Kernel sowie Device Tree und initrd werden mit NVIDIA-eigenen Tools (z. B. tegraflash.py) signiert.
- Optional kann eine Chain-of-Trust mit dem Trusted Execution Environment (TEE) und dem Secure OS (OP-TEE) aufgesetzt werden.
- Der TrustZone-Mechanismus von ARM schützt Schlüsselmaterial und signaturrelevante Operationen vor Zugriff aus dem Linux-Userland.

ESP32-Nodes (Base & Arm)

- Da ESP32 keine native Secure Boot Unterstützung mit Secure Elements in der eingesetzten Konfiguration hat, sollte mindestens ESP-IDF Secure Boot v2 aktiviert werden:
    - Aktiviert SHA-256-basierte Signaturverifikation für die Firmware.
    - Verwendung eines einprogrammierten RSA-Public-Keys (eFuse).
    - Firmware-Updates über OTA werden ebenfalls signaturgeprüft.
    - Aktivierung der Flash Encryption schützt das gespeicherte Firmware-Image zusätzlich vor Auslesen.

Schlüsselmanagement

- Alle kryptographischen Schlüssel werden in einer zentralen, dokumentierten Schlüsselverwaltung verwaltet.
- OEM- und Plattform-Schlüssel werden differenziert:
    - Plattform-Schlüssel (interne Signaturen, Updates)
    - Hersteller-Schlüssel (Secure Boot Root of Trust)
- Schlüsselmaterial wird je nach System in Hardware (UEFI NVRAM, TPM, eFuses) oder dedizierter Infrastruktur (z. B. Air-Gap-Rechner für Signaturen) geschützt.

Tests am Test-Roboter sollten nach jedem Schritt durchgeführt und dokumentiert werden.

4. Umsetzungsempfehlungen – Ubuntu PC

Hauptrechner (Intel i5 mit Q670EI-IM-A)

Der Secure Boot des Motherboards basiert auf UEFI mit aktivierter Secure-Boot-Funktion.

Ziel:

Nur signierte, vom Hersteller oder Systemintegrator geprüfte Bootloader und Kernel dürfen ausgeführt werden.

Maßnahmen:

1. **UEFI Secure Boot aktivieren** (im BIOS/UEFI):
    - BIOS/UEFI → *Boot Configuration* → Secure Boot: Enabled
    - Modus: Standard oder Custom mit eigenem Platform Key (PK), Key Exchange Key (KEK)
2. **Signierte Boot-Komponenten nutzen** (GRUB, Kernel, initrd):
    - Canonical signiert Standard- oder eigener Kernel für Ubuntu Secure Boot.
3. **Status prüfen:**

bash

mokutil --sb-state

Ausgabe: SecureBoot enabled

Custom Keys verwalten:

sudo mokutil --import my_signing_key.der

Damit wird dein eigener Schlüssel ins UEFI MOK (Machine Owner Key) geladen.

Umsetzungsbeschreibung:

Prüfung und Umsetzungsbeschreibung

Bootloader absichern (nur bei GRUB2 Einsatz)

Ziel:

Schutz vor unautorisierten Änderungen am GRUB-Bootmenü.

Maßnahmen:

1. **GRUB-Konfigurationsdatei schützen**:

bash

sudo chmod 000 /boot/grub/grub.cfg

sudo chattr +i /boot/grub/grub.cfg

Anmerkungen: Ein Passwortschutz auf den Bootloader würde zu einer Passworteingabe beim Booten führen und wird nicht weiter betrachtet.

Umsetzungsbeschreibung:

Prüfung und Umsetzungsbeschreibung

Kernel & Root-FS absichern (Integrity Protection)

Ziel:

Verhinderung von Rootkit-Installation durch Manipulation am Kernel oder Dateisystem.

Maßnahmen:

a. dm-verity einrichten (wenn read-only Root möglich):

- Nur sinnvoll bei stabiler Root-FS (kein Logging etc. im Root).
- Nutzung von Tools wie veritysetup:

bash

sudo apt install veritysetup

Alternativ: fs-verity für einzelne Dateien:

bash

sudo apt install fsverity

sudo fsverity enable /usr/bin/critical_binary

Kernel Signing prüfen:

Signatur prüfen:

kmodsign --list-signed /boot/vmlinuz-$(uname -r)

Umsetzungsbeschreibung:

Prüfung und Umsetzungsbeschreibung

Paket- und Softwarehärtung
Ziel:

Nur geprüfte Softwarepakete aus vertrauenswürdigen Quellen installieren.

Weitere Härtungsmaßnahmen (u.a. Deaktivierung nicht benötigter Services folgen in einer separaten Anlage

Maßnahmen:

1. **Ausschließlich signierte Repositories verwenden:**

bash

sudo apt-key list

apt-cache policy

2. **Unbenutzte Paketquellen deaktivieren (z. B. universe, PPA):**

bash

sudo add-apt-repository --remove ppa:xyz

3. **Paketquellen härten mit unattended-upgrades:**

bash

sudo apt install unattended-upgrades

sudo dpkg-reconfigure --priority=low unattended-upgrades

Umsetzungsbeschreibung:

Prüfung und Umsetzungsbeschreibung

ROS-spezifische Sicherheit (Basis)

ROS Noetic kommuniziert per unverschlüsseltem TCP/UDP über ROS-Bridge / roslibjs. Dies ist ein Risiko für Man-in-the-Middle (MITM) und Session Hijacking.

Empfehlungen:

- Isolierung der ROS Bridge in einem separaten Netz (VLAN/Bridge)
- Proxy oder VPN-Gateway nutzen, um Zugriff zu tunneln
- Authentifizierung auf Anwendungsebene (z. B. per Token in App)

Umsetzungsbeschreibung:

Prüfung und Umsetzungsbeschreibung

5. Umsetzungsempfehlungen – Jetson Orin

Secure Boot ist ein Sicherheitsmechanismus, der sicherstellt, dass nur von vertrauenswürdigen Quellen signierte Software während des Bootvorgangs eines Systems ausgeführt wird. Dies verhindert das Laden und Ausführen von bösartigem Code, der das System kompromittieren könnte. Auf dem NVIDIA Jetson Orin Nano Dev Kit bietet JetPack 5.1.3 die notwendigen Tools und Frameworks, um Secure Boot zu implementieren.

Secure-Boot Jetson Orin

Diese Dokumentation führt Sie durch die Konzepte, Voraussetzungen und Schritte zur Aktivierung und Konfiguration von Secure Boot auf Ihrem Jetson Orin Nano Dev Kit.

Der Secure Boot-Prozess auf dem Jetson basiert auf kryptografischen Signaturen und einer Vertrauenskette. Die wichtigsten Komponenten sind:

Boot ROM: Der erste Code, der beim Einschalten des Jetson ausgeführt wird. Er ist in der Hardware fest verdrahtet und unveränderlich. Der Boot ROM überprüft die Signatur des nachfolgenden Bootloaders (UEFI).

UEFI (Unified Extensible Firmware Interface): Ein moderner Bootloader, der die Hardware initialisiert und das Betriebssystem lädt. Bei Secure Boot überprüft UEFI die Signaturen des Kernels, der Kernelmodule und des Root-Dateisystems.

Kryptografische Schlüssel: Zum Signieren von Bootkomponenten werden asymmetrische Kryptographie-Schlüsselpaare (öffentlich/privat) verwendet. Der öffentliche Schlüssel wird im System gespeichert (z. B. in der UEFI-Firmware), während der private Schlüssel sicher aufbewahrt wird und zum Signieren der Software dient.

Vertrauenskette (Chain of Trust): Jede Stufe des Bootprozesses überprüft die Signatur der nächsten Stufe. Der Boot ROM vertraut dem signierten UEFI, UEFI vertraut dem signierten Kernel usw. Dadurch wird sichergestellt, dass nur vertrauenswürdige Software ausgeführt wird.

Voraussetzungen

Bevor Sie mit der Aktivierung von Secure Boot beginnen, stellen Sie sicher, dassfolgende Voraussetzungen erfüllt werden:

- NVIDIA Jetson Orin Nano Dev Kit: Das Zielgerät.
- Host-Computer: Ein Linux-basierter Host-Computer (Ubuntu wird empfohlen) mit installierten NVIDIA SDK Manager-Komponenten.
- JetPack 5.1.3: Auf dem Jetson Orin Nano Dev Kit installiert.
- NVIDIA SDK Manager: Auf dem Host-Computer installiert. Sie benötigen ihn, um die notwendigen Secure Boot-Tools und -Pakete herunterzuladen und Firmware-Images zu flashen.
- Sichere Umgebung für die Schlüsselverwaltung: Die privaten Schlüssel, die zum Signieren der Bootkomponenten verwendet werden, müssen sicher generiert und verwaltet werden. Ein Hardware-Sicherheitsmodul (HSM) wird für Produktionsumgebungen dringend empfohlen. Für Entwicklungszwecke können Software-basierte Lösungen verwendet werden, aber Vorsicht ist geboten.

Generieren von Secure Boot-Schlüsseln

Der erste Schritt besteht darin, kryptografische Schlüsselpaare für Secure Boot zu generieren. Es ist entscheidend, diese Schlüssel sicher zu generieren und zu verwalten.

Erstellen des Arbeitsverzeichnis auf Ihrem Host-Computer:

Bash
mkdir ~/secure_boot cd ~/secure_boot

Generieren vom Geräte-Keyset (Device Keyset): Dieser Keyset wird verwendet, um die Bootloader-Komponenten (UEFI) zu signieren.

Bash
openssl genrsa -out eks.key 4096

openssl rsa -in eks.key -pubout -out eks.pub

Anmerkungen

eks.key: Privater Schlüssel für den Geräte-Keyset.

eks.pub: Öffentlicher Schlüssel für den Geräte-Keyset.

Generieren des Image-Keyset (Image Keyset): Dieser Keyset wird verwendet, um Kernel, Kernelmodule und das Root-Dateisystem zu signieren.

Bash
openssl genrsa -out kernel.key 4096.

openssl rsa -in kernel.key -pubout -out kernel.pub

kernel.key: Privater Schlüssel für den Image-Keyset.

kernel.pub: Öffentlicher Schlüssel für den Image-Keyset.

Hinweis: Die privaten Schlüssel (eks.key, kernel.key) an einem sicheren Ort aufbewahren! Der Verlust dieser Schlüssel würde bedeuten, dass man keine neuen signierten Boot-Images erstellen kann.

Konfigurieren von Secure Boot mit dem NVIDIA SDK Manager

Der NVIDIA SDK Manager bietet eine bequeme Möglichkeit, Secure Boot auf Ihrem Jetson Orin Nano Dev Kit zu konfigurieren und zu aktivieren.

- Starten des NVIDIA SDK Manager auf Ihrem Host-Computer.
- Verbinden des Jetson Orin Nano Dev Kit im Recovery-Modus mit Ihrem Host-Computer. Anweisungen zum Versetzen des Geräts in den Recovery-Modus finden der NVIDIA-Dokumentation.
- Auswahl Zielgerät (Jetson Orin Nano) und JetPack 5.1.3 im SDK Manager aus.
- Navigieren durch die Installationsschritte. Im Schritt "Additional SDKs" oder einer ähnlichen Konfigurationsphase sollten die Optionen für Secure Boot zu finden sein.

Konfigurieren des Secure Boot:

- Der SDK-Manager fordert auf, die öffentlichen Schlüssel (eks.pub, kernel.pub) anzugeben, die im vorherigen Schritt generiert wurden. Pfadangabe der Dateien.
- Möglicherweise gibt es weitere Secure Boot-Konfigurationsoptionen, wie z. B. die Auswahl des Signaturalgorithmus. Sicherheitsanforderungen auswählen.

Flashen des Systems:

Nachdem die Secure Boot-Konfiguration abgeschlossen ist, startet der SDK Manager den Flash-Vorgang. Dieser Prozess installiert die Secure Boot-fähige Firmware und die signierten Bootkomponenten auf Ihrem Jetson Orin Nano Dev Kit.

Manuelle Konfiguration (Alternative zum SDK Manager)

- Alternativ zur Verwendung des SDK Managers kann man Secure Boot auch manuell konfigurieren. Dieser Prozess ist komplexer und erfordert ein tieferes Verständnis der Boot-Pipeline. Hier sind die allgemeinen Schritte:
- Erstellen des angepassten Bootloader-Paket: Dies beinhaltet das Einbinden Ihrer öffentlichen Schlüssel in die UEFI-Firmware. Die genauen Schritte hängen von den NVIDIA-Tools und der Bootloader-Konfiguration ab. (Sie müssen möglicherweise Tools wie mkimage oder ähnliche verwenden).
- Signieren vom Kernel und Root-Dateisystem: Dabei bitte Verwendung des privaten Schlüssel (kernel.key), um den Kernel (Image), die Device Tree Blobs (*.dtb) und optional das Root-Dateisystem zu signieren. NVIDIA bietet in der Regel Signierungstools oder Skripte an.
- Erstellen vom ein Flash-Image: Die signierten Bootloader-Komponenten, den signierten Kernel, die signierten DTBs und das Root-Dateisystem muss in das Flash-fähiges Image gepackt werden.
- Flashen des Image auf das Jetson Orin Nano Dev Kit: Verwenden des NVIDIA-Flash-Tools (z. B. flash.sh) im Recovery-Modus, um das erstellte Image auf das Gerät zu flashen.
- Die detaillierten Schritte für die manuelle Konfiguration sind sehr spezifisch für die JetPack-Version und die verwendeten Tools. Siehe auch: offizielle NVIDIA-Dokumentation für JetPack 5.1.3 für genaue Anweisungen (https://docs.nvidia.com/jetson/jetpack/5.1.3/release-notes/index.html)

Überprüfen von Secure Boot

- Nachdem Secure Boot aktiviert wurde, kann überprüft werden, ob es ordnungsgemäß funktioniert:
- Starten vom Jetson Orin Nano Dev Kit neu.
- Wenn Secure Boot aktiv ist, sollte der Bootloader Signaturen überprüfen. Jegliche unsignierte oder manipulierte Bootkomponente sollte den Bootvorgang unterbrechen und eine Fehlermeldung anzeigen.
- Überprüfen der Boot-Protokolle (falls zugänglich). Die Boot-Protokolle können Informationen über die Secure Boot-Überprüfungen enthalten.
- Es könnte sinnvoll sein, einmal unsignierte Software zu booten (nur zum Test). Wenn Secure Boot ordnungsgemäß funktioniert, sollte das System das Booten von nicht signierter Software ablehnen.

Sichere Schlüsselverwaltung

- Die Sicherheit Ihres Secure Boot-Systems hängt maßgeblich von der sicheren Verwaltung Ihrer privaten Schlüssel ab.
- Generieren der Schlüssel in einer sicheren Umgebung: Dazu idealerweise vertrauenswürdige Computer verwenden (ggf. HSM/TPM) für die Schlüsselgenerierung.
- Beschränken des Zugriffs auf private Schlüssel: Nur autorisierte Personen und Prozesse sollten Zugriff auf die privaten Schlüssel haben.
- Verwenden von starken Passphrasen: Schützen der privaten Schlüssel mit starken Passphrasen!insb. wenn nicht im HSM.
- Erwägen eines HSMs für Produktionsumgebungen: HSMs bieten einen manipulationssicheren Speicher für kryptografische Schlüssel und führen kryptografische Operationen innerhalb der Hardware durch. Ist HSM eine Option??
- Implementieren der Richtlinien für die Schlüsselrotation und -sicherung.
- Alternativ können Offline CAs verwendet werden

Fehlerbehebung

Anbei eine Auflistung möglicher Probleme und deren mögliche Lösungen:

- Booten schlägt fehl nach der Aktivierung von Secure Boot: Dies deutet normalerweise auf ein Problem mit den Signaturen hin. Es muss sichergestellt werden, dass alle Bootkomponenten mit den entsprechenden privaten Schlüsseln signiert wurden und dass die entsprechenden öffentlichen Schlüssel korrekt im System hinterlegt sind.
- Falsche Schlüssel hochgeladen: Die korrekten öffentlichen Schlüssel während der Konfiguration mit dem SDK Manager oder manuell müssen angegeben worden sein – bitte prüfen.
- Inkompatible Softwareversionen: Stellen Sie sicher, dass alle Softwarekomponenten (Bootloader, Kernel, Module) mit der Secure Boot-Implementierung von JetPack 5.1.3 kompatibel sind.

Im weiteren Verlauf werden Details zur Umsetzung dargestellt:

Generierung der Secure Boot-Schlüssel

Ziel:

Erstellung von zwei Schlüsselpaare:

- **EKS (Encryption Key for Secure Bootloader)** für UEFI
- **Kernel Signing Keys** für Kernel & Device Tree

Kommandos

bash

```
# Arbeitsverzeichnis erstellen
mkdir -p ~/secure_boot
cd ~/secure_boot

# EKS (für Bootloader)
openssl genrsa -out eks.key 4096
openssl rsa -in eks.key -pubout -out eks.pub

# Kernel-Signaturschlüssel
openssl genrsa -out kernel.key 4096
openssl rsa -in kernel.key -pubout -out kernel.pub
```

Hinweis: eks.key und kernel.key unbedingt sicher speichern, idealerweise offline oder im HSM!

Aktivierung von Secure Boot mit NVIDIA SDK Manager

Ziel:

Integration der öffentlichen Schlüssel in den Flash-Prozess, Signaturprüfung aktivieren.

Schritte (GUI-basiert):

Jetson im **Recovery-Modus** an den Host anschließen (lsusb zeigt NVIDIA-Gerät)

NVIDIA **SDK Manager** starten → Jetson Orin Nano auswählen

JetPack 5.1.3 auswählen

1. Im Abschnitt „Secure Boot / Key Configuration“:
    - eks.pub und kernel.pub einfügen
    - Signaturalgorithmen wählen (Standard: RSA-2048 oder RSA-4096)
2. Flash-Vorgang starten

Der SDK-Manager integriert die Keys, aktiviert Secure Boot und erstellt signierte Images.

Manuelle Signierung

Ziel:

Mehr Kontrolle oder Automatisierung ohne SDK GUI.

Typische Schritte:

bash

Beispiel: Signieren eines Kernel-Images (Image.gz)

openssl dgst -sha512 -sign kernel.key -out Image.sig Image

Optional: DTB signieren

openssl dgst -sha512 -sign kernel.key -out tegra.dtb.sig tegra.dtb

Danach:

- Signierte Dateien in das **Linux-for-Tegra (L4T)** Directory integrieren
- Flashen mit:

bash

cd ~/nvidia/nvidia_sdk/JetPack_5.1.3/Linux_for_Tegra

sudo ./flash.sh jetson-orin-nano-devkit mmcblk0p1

Überprüfung nach Aktivierung

Ziel:

Validieren, dass Secure Boot korrekt arbeitet.

Schritte:

- Gerät booten lassen
- Unsignierte Kernel/Bootloader testweise einspielen → Bootfehler sollte auftreten
- Logfiles aus UART oder /var/log/boot.log auslesen

Alternativ:

bash

journalctl -k | grep -i secure

Empfehlungen für Schlüsselmanagement

Maßnahme	**Beschreibung**
Schlüssel mit Passphrase sichern	z. B. openssl genrsa -aes256
Zugriff beschränken	Nur autorisierte Nutzer/Gruppen
Schlüsselrotation planen	Bei Schlüsselverlust oder -kompromittierung
Dev / Prod trennen	Unterschiedliche Schlüssel für Test und Produktion
Optional: HSM	Hardware-gesicherte Signaturprozesse nutzen

Fehlerquellen & Troubleshooting

Problem	Mögliche Ursache
Bootfehler nach Flash	Kernel oder DTB nicht signiert
Unsichere Images starten trotzdem	Secure Boot nicht aktiv, Keys fehlen
Signaturprüfung schlägt fehl	Schlüssel nicht korrekt eingebunden (falsches Format?)

Umsetzungsbeschreibung:

Prüfung und Umsetzungsbeschreibung

Anlagen