CAN(Open) Schnittstellen - Sicherheit

Teilsicherheitskonzept – CONBOTICS MalerRoboter

Inhalt

1.) Analyse der Sicherheitsrisiken des CAN-Busses und Strategien zur Absicherung in modernen Systemen

1. Einleitung

Der Controller Area Network (CAN)-Bus ist ein robustes und kosteneffizientes serielles Bussystem, das in den 1980er Jahren für die Vernetzung von Steuergeräten (ECUs) in Kraftfahrzeugen entwickelt wurde. Seine Zuverlässigkeit und Echtzeitfähigkeit haben zu seiner weiten Verbreitung geführt, nicht nur in der Automobilindustrie, sondern auch in der Industrieautomation, der Medizintechnik und der Luft- und Raumfahrt.

Jedoch wurde der CAN-Bus in einer Zeit entworfen, in der Cybersicherheit keine Rolle spielte. Das ursprüngliche Protokolldesign konzentrierte sich ausschließlich auf die Robustheit der Signalübertragung und die Fehlererkennung, nicht auf den Schutz vor böswilligen Angriffen. Mit der zunehmenden Vernetzung von Fahrzeugen und Industrieanlagen mit externen Netzwerken (z. B. dem Internet über Telematikeinheiten) sind die internen Bussysteme zu einem primären Ziel für Cyberangriffe geworden.

Dieser Bericht analysiert die inhärenten Sicherheitslücken des CAN-Protokolls, beschreibt die daraus resultierenden Angriffsvektoren und stellt ein mehrschichtiges Sicherheitskonzept zur wirksamen Absicherung von CAN-Schnittstellen vor.

2. Inhärente Sicherheitslücken des CAN-Protokolls

Die grundlegenden Schwachstellen des CAN-Busses resultieren direkt aus seinem Design, das auf Vertrauen und Einfachheit basiert:

Keine Authentifizierung: Das CAN-Protokoll sieht keine Methode vor, um die Identität eines Senders zu überprüfen. Jedes an den Bus angeschlossene Steuergerät kann Nachrichten senden, und die Empfänger haben keine Möglichkeit zu verifizieren, ob eine Nachricht von der erwarteten Quelle stammt. Dies öffnet Tür und Tor für die Fälschung von Nachrichten (Spoofing).

Fehlende Verschlüsselung: Die auf dem CAN-Bus übertragenen Daten (CAN-Frames) sind unverschlüsselt und für jedes lauschende Gerät im Klartext lesbar. Sensible Steuerbefehle oder Diagnosedaten können somit leicht abgehört werden.

Broadcast-Kommunikation: Nachrichten werden an alle Teilnehmer des Busses gesendet (Broadcast). Es gibt kein Konzept für eine direkte, private Kommunikation zwischen zwei Steuergeräten.

Keine Integritätsprüfung auf Nachrichtenebene: Obwohl ein Cyclic Redundancy Check (CRC) zur Erkennung von Übertragungsfehlern dient, gibt es keinen kryptografischen Mechanismus (wie einen Message Authentication Code, MAC), um sicherzustellen, dass eine Nachricht während der Übertragung nicht absichtlich manipuliert wurde.

Prioritätsbasierte Arbitrierung: Der Zugriff auf den Bus wird über die Nachrichten-ID geregelt. Nachrichten mit einer niedrigeren ID haben eine höhere Priorität. Ein Angreifer kann den Bus lahmlegen, indem er ihn kontinuierlich mit Nachrichten höchster Priorität flutet.

3. Gängige Angriffsvektoren und Bedrohungsszenarien

Angreifer können diese Schwachstellen über verschiedene Vektoren ausnutzen, um die Kontrolle über Fahrzeug- oder Systemfunktionen zu erlangen.

Physische Angriffsvektoren:

OBD-II-Port (On-Board Diagnostics): Der standardisierte Diagnoseport in Fahrzeugen bietet direkten Zugriff auf den CAN-Bus. Angreifer können über diesen Port bösartige Geräte anschließen, um Nachrichten zu senden oder den Busverkehr zu analysieren.

Kompromittierung von Steuergeräten: Der direkte Anschluss an Kabelbäume nach dem Entfernen von Verkleidungen ermöglicht ebenfalls den Zugriff. Die sogenannte **"CAN-Injection"**, bei der Diebe sich Zugang zum Scheinwerferkabelbaum verschaffen, um das Fahrzeug zu entriegeln und zu starten, ist ein praxisrelevantes Beispiel.

Remote-Angriffsvektoren:

Kompromittierte Infotainment-Systeme: Systeme mit Bluetooth-, WLAN- oder Mobilfunkverbindung sind ein Hauptziel. Gelingt es einem Angreifer, das Infotainment-System zu kompromittieren, kann er dieses als Brücke nutzen, um auf kritischere CAN-Busse (z. B. den Antriebs- oder Fahrwerks-Bus) zuzugreifen.

Telematikeinheiten (TCU): Diese für "Connected Car"-Dienste zuständigen Einheiten sind per Definition mit dem Internet verbunden und stellen ein potenzielles Gateway für Fernangriffe dar.

Sensoren und Peripherie: Auch scheinbar harmlose Komponenten wie Reifendrucksensoren oder Funkschlüssel können als Einstiegspunkt dienen.

Typische Angriffe auf den CAN-Bus:

Denial-of-Service (DoS)-Angriff: Ein Angreifer flutet den Bus mit Nachrichten hoher Priorität, sodass reguläre, kritische Nachrichten (z. B. vom ABS-Steuergerät) nicht mehr durchkommen. Dies kann zum Ausfall von Sicherheitsfunktionen führen.

Spoofing-Angriff (Nachrichtenfälschung): Ein Angreifer sendet gefälschte Nachrichten, um unerwünschte Aktionen auszulösen. Beispiele sind das Entriegeln der Türen, das Deaktivieren der Wegfahrsperre oder das Senden gefälschter Sensordaten (z. B. eine falsche Geschwindigkeit an das Getriebesteuergerät).

Replay-Angriff: Ein Angreifer zeichnet eine legitime Kommunikation (z. B. den Befehl zum Öffnen der Tür per Funkschlüssel) auf und spielt sie zu einem späteren Zeitpunkt erneut ab, um die Aktion zu wiederholen.

Bus-Off-Angriff: Durch das gezielte Erzeugen von Fehler-Frames kann ein Angreifer ein bestimmtes Steuergerät dazu zwingen, sich aufgrund interner Fehlerzähler vom Bus zu trennen und abzuschalten.

4. Mehrschichtige Sicherheitsstrategien (Defense in Depth)

Da das CAN-Protokoll selbst nicht geändert werden kann, muss die Sicherheit durch zusätzliche, überlagerte Mechanismen gewährleistet werden.

4.1. Netzwerksegmentierung und Gateways

Moderne Fahrzeug- und Systemarchitekturen segmentieren das Netzwerk. Verschiedene CAN-Busse werden für unterschiedliche Domänen (z. B. Infotainment, Antrieb, Komfort) verwendet und über ein zentrales Gateway miteinander verbunden. Dieses Gateway agiert als Firewall und kontrolliert den Nachrichtenfluss zwischen den Segmenten. Es lässt nur definierte und plausible Nachrichten von einem Bus zum anderen passieren und blockiert unautorisierte Kommunikationsversuche.

4.2. Intrusion Detection und Prevention Systeme (IDS/IPS)

Ein IDS überwacht den CAN-Bus-Verkehr in Echtzeit auf Anomalien und bekannte Angriffsmuster. Man unterscheidet hauptsächlich zwei Ansätze:

Signaturbasierte Erkennung: Das IDS sucht nach bekannten Mustern bösartiger Nachrichten (ähnlich wie ein Virenscanner).

Anomaliebasierte Erkennung: Das IDS lernt während einer Anlernphase den normalen Busverkehr (z. B. welche Nachrichten-IDs wie oft auftreten). Jede signifikante Abweichung von diesem Normalzustand, wie eine ungewöhnlich hohe Frequenz einer bestimmten Nachricht, wird als potenzieller Angriff gemeldet. Hier kommen zunehmend auch Machine-Learning-Algorithmen zum Einsatz. Ein IPS kann zusätzlich aktiv eingreifen und bösartige Nachrichten blockieren.

4.3. Kryptografische Absicherung (Verschlüsselung und Authentifizierung)

Um die Schwächen des Protokolls direkt zu adressieren, werden kryptografische Verfahren über dem CAN-Standard implementiert.

Message Authentication Codes (MACs): Jede kritische Nachricht wird um einen kryptografischen Anhang (den MAC) erweitert, der mit einem geheimen Schlüssel berechnet wird. Der Empfänger kann mit demselben Schlüssel überprüfen, ob die Nachricht authentisch und unverändert ist. Ansätze wie **SecOC (Secure On-Board Communication)** im AUTOSAR-Standard definieren diesen Mechanismus.

Verschlüsselung: Obwohl aufgrund der begrenzten Datenkapazität von CAN-Frames (8 Bytes bei Classic CAN, 64 bei CAN FD) und den Echtzeitanforderungen schwierig, kann eine teilweise oder vollständige Verschlüsselung für besonders sensible Daten angewendet werden.

Sicheres Schlüsselmanagement: Die sichere Speicherung und Verteilung der kryptografischen Schlüssel ist entscheidend. Dies erfolgt oft über dedizierte **Hardware Security Modules (HSMs)**, die in den Steuergeräten integriert sind.

4.4. Einhaltung von Standards und Regularien

Neue gesetzliche Vorschriften und Industriestandards zwingen Hersteller zur Implementierung von Cybersicherheitsmaßnahmen.

ISO/SAE 21434: Dieser internationale Standard definiert einen umfassenden Rahmen für das "Cybersecurity Engineering" in Straßenfahrzeugen. Er fordert einen risikobasierten Ansatz über den gesamten Lebenszyklus des Fahrzeugs – von der Entwicklung über die Produktion bis hin zur Stilllegung.

UN-Regulierung Nr. 155 (UN R155): Diese für die Typgenehmigung von Fahrzeugen verbindliche Regelung fordert von Herstellern die Implementierung eines zertifizierten Cybersecurity Management Systems (CSMS) und dessen Anwendung auf die Fahrzeugarchitektur.

5. Fazit und Handlungsempfehlung

Die Schnittstellensicherheit des CAN-Busses ist eine kritische Herausforderung für alle Branchen, die dieses Protokoll einsetzen. Die inhärenten Sicherheitslücken können nicht ignoriert werden. Ein wirksamer Schutz kann nur durch einen ganzheitlichen **"Defense in Depth"-Ansatz** erreicht werden, der präventive, detektivische und reaktive Maßnahmen kombiniert.

Für Unternehmen und Entwickler sind folgende Schritte unerlässlich:

Risikoanalyse durchführen: Identifizieren Sie kritische Systeme und potenzielle Angriffsvektoren gemäß Standards wie ISO/SAE 21434.

Netzwerkarchitektur sichern: Implementieren Sie eine strikte Segmentierung durch sichere Gateways, um Angriffsflächen zu minimieren.

Verkehr überwachen: Setzen Sie Intrusion Detection Systeme ein, um Angriffe in Echtzeit zu erkennen.

Kommunikation härten: Führen Sie Message Authentication (z. B. mittels SecOC) für alle sicherheitskritischen Kommunikationen ein.

Zugänge schützen: Sichern Sie alle physischen und drahtlosen Schnittstellen (OBD-Port, Bluetooth, WLAN) rigoros ab.

Nur durch die konsequente Umsetzung dieser mehrschichtigen Strategie kann die Integrität und Sicherheit von CAN-basierten Systemen in der vernetzten Welt von heute und morgen gewährleistet werden.

2.) Testen der CAN Bus Schnitstelle unter Linux:

Das Testen von CAN-Schnittstellen unter Linux ist dank der nativen Unterstützung im Kernel (bekannt als **SocketCAN**) und einer leistungsstarken Sammlung von Kommandozeilen-Tools sehr direkt und effektiv möglich. Der Standard-Werkzeugkasten dafür ist **can-utils**.

Dieser Leitfaden erklärt, wie Sie eine CAN-Schnittstelle auf einem Linux-System (z. B. der Steuerungseinheit eines Roboters) einrichten und mit den can-utils testen.

1.) Voraussetzung: Installation von can-utils

Stellen Sie zunächst sicher, dass die can-utils installiert sind. Auf den meisten Debian-basierten Systemen (wie Ubuntu oder Raspberry Pi OS) geht das einfach über den Paketmanager:

Bash

```
sudo apt update
```

```
sudo apt install can-utils
```

Schritt 1: CAN-Schnittstelle einrichten und aktivieren

Bevor Sie testen können, muss die CAN-Schnittstelle konfiguriert und "hochgefahren" werden, ähnlich wie eine normale Netzwerkschnittstelle (z. B. eth0).

Möglichkeit A: Physische CAN-Schnittstelle (z.B. USB-to-CAN-Adapter)

Wenn Sie einen echten CAN-Adapter verwenden (z. B. von PEAK-System, Kvaser oder ein günstigerer slcan-Adapter), wird dieser normalerweise als can0, can1 usw. im System erkannt.

1. **Schnittstelle konfigurieren und aktivieren:** Sie müssen die Bitrate passend zum CAN-Bus des Roboters einstellen. Wenn die Bitrate beispielsweise 500 kbit/s beträgt, lautet der Befehl:

Bash

```
# Setzt den Typ der Schnittstelle auf "can" und konfiguriert die Bitrate
sudo ip link set can0 type can bitrate 500000


# Fährt die Schnittstelle hoch
```

```
sudo ip link set up can0
```

2. **Schnittstellen-Status prüfen:** Mit `ip -details link show can0` können Sie den Status und die Konfiguration überprüfen. Sie sollten sehen, dass der Status auf "UP" steht.

Möglichkeit B: Virtuelle CAN-Schnittstelle (zum Üben und für Simulation)

Wenn Sie keinen physischen Zugang zum Bus haben oder Software testen möchten, ohne echte Hardware zu beeinflussen, können Sie eine virtuelle CAN-Schnittstelle (`vcan`) erstellen.

1. **Kernel-Modul laden:**

```
Bash

sudo modprobe vcan
```

2. **Virtuelle Schnittstelle erstellen und aktivieren:**

```
Bash

sudo ip link add dev vcan0 type vcan

sudo ip link set up vcan0
```

Jetzt haben Sie eine `vcan0`-Schnittstelle, die sich wie eine echte CAN-Schnittstelle verhält, aber nur innerhalb Ihres Linux-Systems existiert. Das ist perfekt, um die `can-utils` zu erlernen.

Schritt 2: CAN-Bus-Verkehr analysieren und testen

Jetzt, da die Schnittstelle aktiv ist, können Sie mit den eigentlichen Tests beginnen. Öffnen Sie für die folgenden Beispiele am besten zwei Terminals: eines zum Senden und eines zum Empfangen.

Test 1: Verkehr passiv mithören (`candump`)

`candump` ist das "tcpdump" für CAN. Es zeigt alle Nachrichten an, die auf dem Bus gesendet werden.

- **Alle Nachrichten auf `can0` anzeigen:**

```
Bash

candump can0
```

Sie sehen nun eine Live-Ausgabe aller CAN-Nachrichten im Format: <interface> <CAN-ID>#<Daten>

- **Ausgabe farbig und mit ASCII-Zeichen darstellen:**
-

Bash

```
candump -ca can0
```

Dies erleichtert die Lesbarkeit erheblich, da die Daten gleichzeitig hexadezimal und als Text angezeigt werden.

- **Nachrichten filtern:** Um nur Nachrichten mit einer bestimmten CAN-ID (z. B. 1A4) anzuzeigen:

Bash

```
candump can0,1A4:1FFFFFFF
```

(Die Maske 1FFFFFFF sorgt für eine exakte Übereinstimmung der ID).

Test 2: Einzelne CAN-Nachricht senden (cansend)

Mit cansend können Sie eine einzelne, selbst definierte CAN-Nachricht auf den Bus senden. Dies ist der Kern vieler Penetrationstests, um die Reaktion des Roboters auf unerwartete Befehle zu testen.

- **Senden einer Nachricht:** Der Befehl sendet eine Nachricht mit der CAN-ID 123 und den 8 Datenbytes 11223344AABBCCDD.

Bash

```
cansend can0 123#11223344AABBCCDD
```

Wenn Sie candump in einem anderen Terminal laufen haben, werden Sie diese Nachricht dort sofort sehen.

- **Test-Szenario für einen Roboter:** Wenn Sie durch vorherige Analyse (Reverse Engineering) wissen, dass die ID 250 mit den Daten 01000000 den Greifer des Roboters öffnet, könnten Sie diesen Befehl gezielt senden, um zu sehen, ob das System den Befehl ohne weitere Authentifizierung akzeptiert:

Bash

```
cansend can0 250#01000000
```

Test 3: Zufälligen Verkehr generieren (cangen)

cangen (CAN-Generator) wird verwendet, um den Bus unter Last zu setzen oder um zu sehen, wie das System auf zufällige, unbekannte Nachrichten reagiert.

- **Zufällige Nachrichten generieren:**

```
Bash

cangen can0
```

Dies erzeugt einen kontinuierlichen Strom von CAN-Nachrichten mit zufälligen IDs und Daten.

- **Denial-of-Service-Angriff simulieren:** Ein Angreifer könnte versuchen, den Bus mit Nachrichten höchster Priorität (niedrigste ID) zu fluten. Dies können Sie simulieren, um zu sehen, ob die reguläre Kommunikation des Roboters gestört wird:

```
Bash

# Generiert nur Nachrichten mit ID 0

cangen can0 -i 0 -L 8 -D i
```

Test 4: Verkehr aufzeichnen und wieder abspielen (canplayer)

Dies ist nützlich für Replay-Angriffe.

1. **Verkehr aufzeichnen:** Zeichnen Sie den gesamten Verkehr in eine Log-Datei auf, während Sie den Roboter eine bestimmte Aktion ausführen lassen (z. B. eine programmierte Bewegung).

```
Bash

candump -l can0
```

Dies erstellt eine Datei namens candump-YYYY-MM-DD_HHMMSS.log.

2. **Verkehr wieder abspielen:** Spielen Sie die aufgezeichnete Sequenz erneut auf dem Bus ab, um zu sehen, ob der Roboter die gleiche Aktion wiederholt.

```
Bash

canplayer -I candump-YYYY-MM-DD_HHMMSS.log
```

Wenn der Roboter die Bewegung wiederholt, haben Sie eine erfolgreiche Replay-Attacke demonstriert.

Zusammenfassung des Test-Workflows

1. **Schnittstelle einrichten:** `ip link set can0 up type can bitrate ...`
2. **Passiv lauschen:** `candump` can0 nutzen, um ein Gefühl für den normalen Verkehr zu bekommen und interessante CAN-IDs zu identifizieren.
3. **Gezielte Befehle senden:** `cansend can0 <ID>#<DATA>` verwenden, um Hypothesen zu testen (z. B. "Steuert ID X die Bewegung Y?").
4. **Stress-Tests durchführen:** `cangen can0` einsetzen, um die Robustheit des Systems gegen Buslast oder zufällige Daten zu prüfen.
5. **Replay-Angriffe testen:** Mit `candump -l` und `canplayer` Aktionen aufzeichnen und wiederholen.

Diese Werkzeuge bieten eine umfassende Grundlage für die Analyse und das sicherheitstechnische Testen von CAN-Schnittstellen direkt unter Linux.

3.) Analyse der Sicherheitsrisiken des CANopen-Protokolls und Strategien zur Absicherung in modernen Systemen

Die Analyse der Sicherheitsrisiken von CANopen und die entsprechenden Absicherungsstrategien baut auf den grundlegenden Schwächen des CAN-Busses auf, erweitert diese aber um die spezifischen Risiken des höheren Protokolls. Ein offener Zugang zu einem CANopen-Netzwerk ist daher noch kritischer als bei einem reinen CAN-Bus.

---

1. Sicherheitsrisiken des CANopen-Protokolls

CANopen erbt alle Risiken des darunterliegenden CAN-Busses (keine Verschlüsselung, keine Authentifizierung) und fügt durch seine standardisierten Dienste neue, spezifische Angriffsvektoren hinzu.

- **Manipulation der Netzwerkverwaltung (NMT)**: Das NMT-Protokoll ist das Herzstück der Gerätesteuerung. Ein Angreifer kann gezielt NMT-Befehle senden, um einzelne oder alle Geräte im Netzwerk zu stören.
    - **Angriff:** Senden eines "Reset Node" oder "Stop Node" Befehls an ein kritisches Gerät (z.B. den Hauptmotor-Controller eines Roboters).
    - **Folge:** Das System oder Teile davon fallen sofort aus, was zu unkontrollierten Zuständen oder einem kompletten Stillstand führen kann.
- **Unerlaubter Zugriff auf das Objektverzeichnis (SDO-Angriffe)**: Jedes CANopen-Gerät besitzt ein Objektverzeichnis (Object Dictionary), das alle Konfigurationsparameter und Prozessdaten enthält. Über SDOs kann dieses Verzeichnis gelesen und beschrieben werden.
    - **Auslesen:** Ein Angreifer kann sensible Konfigurationsdaten, Betriebsparameter oder Herstellerinformationen auslesen.
    - **Schreiben:** Dies ist der gefährlichste Angriff. Ein Angreifer kann kritische Parameter überschreiben, z.B. Sicherheitsgrenzwerte, Kalibrierungsdaten oder die Gerätekonfiguration. Dies kann das Verhalten eines Geräts fundamental und permanent verändern.

- **Fälschung von Echtzeitdaten (PDO-Spoofing)**: PDOs werden für die schnelle Übertragung von Echtzeitdaten wie Sensorwerten oder Steuerbefehlen genutzt.
    - **Angriff:** Ein Angreifer sendet gefälschte PDOs, die z.B. falsche Positionsdaten eines Roboterarms oder einen harmlosen Status eines Sensors melden, obwohl eine Gefahr besteht.
    - **Folge:** Das System trifft Entscheidungen auf Basis falscher Daten, was zu Fehlfunktionen, Kollisionen oder der Umgehung von Sicherheitsmechanismen führen kann.
- **Missbrauch von Sicherheitsmechanismen**: Selbst Schutzfunktionen wie "Heartbeat" (Lebenszeichen) können missbraucht werden. Ein Angreifer kann Heartbeat-Nachrichten für ein Gerät fälschen, das er zuvor per DoS-Angriff lahmgelegt hat, um dem System vorzugaukeln, es sei alles in Ordnung.

2.) Strategien zur Absicherung in modernen Systemen

Die Absicherung eines CANopen-Systems erfordert einen mehrschichtigen Ansatz ("Defense in Depth"), der über die Absicherung des reinen CAN-Busses hinausgeht.

- **Physischer Zugriffsschutz**: Die absolut erste und wichtigste Maßnahme ist, offene Schnittstellen physisch zu blockieren (z.B. mit Port-Blockern) und Steuerungsschränke abzuschließen.
- **CANopen-fähiges Secure Gateway**: Ein Secure Gateway muss hier nicht nur CAN-IDs filtern, sondern auch das CANopen-Protokoll verstehen.
    - **SDO-Filterung:** Das Gateway kann so konfiguriert werden, dass es Schreibzugriffe (SDOs) auf besonders kritische Objekte im Objektverzeichnis blockiert oder nur von einer einzigen, autorisierten Master-Adresse zulässt.
    - **NMT-Filterung:** Das Gateway kann verhindern, dass globale NMT-Befehle (z.B. "Reset all Nodes") von einer nicht autorisierten Quelle durch das Netzwerk gesendet werden.
- **Intrusion Detection System (IDS) mit Protokoll-Verständnis**: Ein modernes IDS für CANopen überwacht den Verkehr auf Protokollebene.
    - **Anomalie-Erkennung:** Das IDS kann Alarm schlagen, wenn z.B. ein SDO-Schreibzugriff auf ein Objekt stattfindet, das normalerweise nur gelesen wird, oder wenn ein Gerät plötzlich aufhört, seinen Heartbeat zu senden.

- **Applikationsinterne Authentifizierung**: Da CANopen selbst keine Authentifizierung bietet, kann diese in der Anwendungssoftware implementiert werden.
    - **Challenge-Response-Verfahren:** Bevor ein kritischer Parameter per SDO geschrieben werden darf, sendet das Gerät eine "Challenge" (eine Zufallszahl) an den Master. Der Master muss diese mit einem geheimen Schlüssel korrekt beantworten ("Response"), bevor der Schreibvorgang akzeptiert wird.
- **Secure Boot und Firmware-Signierung**: Es muss sichergestellt werden, that die auf den Steuergeräten laufende Software authentisch und unverändert ist. Jedes Firmware-Update muss kryptografisch signiert sein, um das Einspielen von manipulierter Software, die z.B. Sicherheitsabfragen umgeht, zu verhindern.

Durch die Kombination dieser architektonischen, softwarebasierten und prozessualen Maßnahmen kann die Angriffsfläche eines offenen CANopen-Systems erheblich reduziert und die Integrität des Gesamtsystems gewährleistet werden.

4.) Testen der CAN Open Schnittstelle unter Linux

Das Testen einer CANopen-Schnittstelle unter Linux geht einen Schritt weiter als das reine Testen des CAN-Busses. Während can-utils die physikalische Schicht testet, benötigst du für CANopen Werkzeuge, die die höhere Protokollebene verstehen.

Der effizienteste Weg, dies unter Linux zu tun, ist die Verwendung der **python-canopen** Bibliothek. Sie ermöglicht es dir, automatisierte Tests für alle wichtigen CANopen-Funktionen zu erstellen.

1.) Voraussetzungen

1. **Hardware:** Du benötigst einen Computer mit Linux und einen USB-to-CAN-Adapter.
2. **Software-Installation:**

Bash

```
# Installiert die CAN-Tools für die Basiseinrichtung
sudo apt update
sudo apt install can-utils

# Installiert die Python-Bibliothek für CANopen-Tests
pip install python-canopen
```

3. **Einrichtung der CAN-Schnittstelle:** Wie bei jedem CAN-Test musst du zuerst die physikalische Schnittstelle mit der korrekten Bitrate konfigurieren und aktivieren. Für eine Bitrate von 250 kbit/s:

Bash

```
sudo ip link set can0 type can bitrate 250000
sudo ip link set up can0
```

2.) Testszenarien für eine CANopen-Schnittstelle

Die folgenden Tests werden als Python-Skripte ausgeführt. Speichere den Code jeweils in einer Datei (z.B. test.py) und führe sie mit python3 test.py aus.

Szenario 1: Netzwerk-Scan (Geräte finden)

Ziel: Alle aktiven CANopen-Geräte (Nodes) im Netzwerk finden und ihre Node-IDs identifizieren.

Beschreibung: Dieser Test sendet eine NMT (Network Management) "Node Guarding"-Anfrage und lauscht auf die Antworten der Slaves.

Python-Skript:

Python

```
import canopen
import time

# Erstellt ein Netzwerk-Objekt und verbindet es mit deiner CAN-Schnittstelle
network = canopen.Network()
network.connect(bustype='socketcan', channel='can0')

print("Suche nach aktiven Geräten...")
# Führt einen Scan des Netzwerks durch
network.scanner.search()
# Warte eine Sekunde, damit alle Geräte antworten können
time.sleep(1)

if not network.scanner.nodes:
    print("Keine Geräte gefunden.")
else:
    print("Gefundene Geräte:")
    for node_id in network.scanner.nodes:
```

```
    print(f"  - Node-ID: {node_id}")


network.disconnect()
```

Szenario 2: Objektverzeichnis lesen/schreiben (SDO-Zugriff)

Ziel: Konfigurationsparameter von einem Gerät lesen und schreiben. Dies testet die SDO (Service Data Object) Kommunikation.

Beschreibung: Fast jedes CANopen-Gerät hat ein standardisiertes Objektverzeichnis. Wir lesen einen Standardparameter (Gerätename) und schreiben einen Konfigurationsparameter (Heartbeat-Zeit).

Python-Skript: (Ersetze NODE_ID durch eine ID, die du in Szenario 1 gefunden hast)

```
Python

import canopen


NODE_ID = 10 # <-- Ersetze dies durch die ID deines Geräts


network = canopen.Network()

network.connect(bustype='socketcan', channel='can0')


# Fügt den bekannten Node zum Netzwerk hinzu

node = network.add_node(NODE_ID, 'path/to/your/device.eds') # Optional: EDS-Datei für Objektnamen


# Test 1: Lesen des Gerätenamens (Objekt 0x1008)

try:

  device_name = node.sdo[0x1008].raw

  print(f"Gerätename von Node {NODE_ID}: {device_name.decode('utf-8')}")

except canopen.sdo.SdoAbortedError as e:
```

```
    print(f"Fehler beim Lesen von Objekt 0x1008: {e}")

# Test 2: Lesen und Schreiben der Heartbeat-Zeit (Objekt 0x1017)
try:
    current_heartbeat_ms = node.sdo[0x1017].raw
    print(f"Aktuelle Heartbeat-Zeit: {current_heartbeat_ms} ms")

    # WICHTIG: Schreibe nur, wenn du weißt, dass es sicher ist!
    print("Setze Heartbeat-Zeit auf 2000 ms...")
    node.sdo[0x1017].raw = 2000 # Setzt die Heartbeat-Zeit auf 2 Sekunden

    new_heartbeat_ms = node.sdo[0x1017].raw
    print(f"Neue Heartbeat-Zeit: {new_heartbeat_ms} ms")

except canopen.sdo.SdoAbortedError as e:
    print(f"Fehler beim Zugriff auf Objekt 0x1017: {e}")

network.disconnect()
```

Szenario 3: Gerätestatus steuern (NMT)

Ziel: Überprüfen, ob das Gerät korrekt auf Befehle zum Ändern des Betriebszustands reagiert.

Beschreibung: Wir versetzen das Gerät in den Zustand `OPERATIONAL` (normaler Betrieb), dann in den Zustand `STOPPED` und führen schließlich einen Reset durch.

Python-Skript: (Ersetze `NODE_ID`)

```
Python
import canopen
```

```
import time
NODE_ID = 10 # <-- Ersetze dies

network = canopen.Network()
network.connect(bustype='socketcan', channel='can0')
node = network.add_node(NODE_ID)

def on_heartbeat(message):
    print(f"Heartbeat von Node {NODE_ID} empfangen. Aktueller Status: {node.nmt.state}")

node.heartbeat.add_callback(on_heartbeat)

print("Setze Node in den OPERATIONAL-Zustand...")
node.nmt.state = 'OPERATIONAL'
time.sleep(2)

print("Setze Node in den STOPPED-Zustand...")
node.nmt.state = 'STOPPED'
time.sleep(2)

print("Führe Node-Reset durch...")
node.nmt.state = 'RESET'
time.sleep(2) # Warte auf den Neustart

print("Test beendet.")
network.disconnect()
```

Szenario 4: Echtzeitdaten und Notfallmeldungen überwachen (PDO & EMCY)

Ziel: Überprüfen, ob das Gerät seine Echtzeitdaten (PDOs) sendet und im Fehlerfall eine Notfallmeldung (EMCY) absetzt.

Beschreibung: Dies ist hauptsächlich ein Test des Zuhörens. Nachdem ein Node in den OPERATIONAL-Zustand versetzt wurde, sollte er automatisch seine PDOs senden.

Testschritte:

1. **PDOs überwachen:** Versetze den Node mit dem Skript aus Szenario 3 in den OPERATIONAL-Zustand. Beobachte parallel in einem zweiten Terminal den rohen CAN-Verkehr mit candump can0. Du solltest nun regelmäßig Nachrichten sehen, deren CAN-IDs den PDOs des Geräts entsprechen.
2. **EMCY testen:**
    - Füge in dein Python-Skript einen "Callback" für Notfallmeldungen ein:

```
Python

def on_emcy(emcy):
    print(f"Notfallmeldung empfangen: {emcy}")


network.emcy.add_callback(on_emcy)
```

    - Lass das Skript laufen und provoziere einen Fehler am Gerät (falls dies sicher möglich ist, z.B. durch Abziehen eines Sensors).
    - Überprüfe, ob die Notfallmeldung in deinem Skript oder in candump erscheint.

Diese Testszenarien decken die grundlegenden und wichtigsten Funktionen des CANopen-Protokolls ab und bieten eine solide Basis für eine umfassende Sicherheits- und Funktionsprüfung deiner Schnittstelle.