PoC on Framework: Automotive Threat Matrix (ATM) 🚙

Overview: The Automotive Threat Matrix defines 13 adversarial tactics, inspired by ATT&CK but tailored to vehicles [Tactics - Enterprise | MITRE ATT&CK®](), [An introduction into the Automotive Threat Matrix | Block Harbor Insights](). While exact tactic names aren't publicly enumerated in our sources, the ATM whitepaper indicates about 13 tactics and ~80 techniques.

We'll proceed on these tactics: ATM

🔐 **#1 — Tactic: Manipulate Environment (ATA-001)**

🎯 **Goal:** Influence or interfere with the vehicle's surrounding environment to degrade system performance or trick sensors. This includes attacks like rogue transmissions, jamming, spoofing, or physically tampering with environmental inputs.

🛰️ **Technique #1: ATE-002 – Rogue Cellular Base Station**

🔹 **Procedure 1:** LTE Base Station Setup & Intercept

1. **Install srsLTE or OpenBTS on a Linux machine with USRP (SDR) hardware.**
    - To begin simulating a rogue cellular base station, install open-source LTE/GSM stack tools like srsLTE (for LTE) or OpenBTS (for GSM) on a Linux system. These tools interface with USRP (Universal Software Radio Peripheral) hardware, which acts as a programmable radio for transmitting and receiving wireless signals. Once installed, the system can mimic a real cellular tower, allowing an attacker to create a fake base station environment and lure nearby vehicle telematics units into connecting to it.
2. **Configure MCC and MNC to match the target region's cellular provider.**
    - Set the Mobile Country Code (MCC) and Mobile Network Code (MNC) in the rogue base station software to match those of the legitimate telecom provider in the region. This ensures that nearby telematics units and ECUs

mistake the fake base station for an authentic one, making it more likely that the vehicle will connect automatically without raising suspicion.

3. **Lower TX power to create a localized fake tower that won't raise alerts.**
    - Reduce the transmission (TX) power of your rogue base station to cover only a small, specific area like a parking lot or garage. This keeps the operation stealthy and avoids interfering with wider legitimate network signals, which could trigger alerts to carriers or affect other devices beyond your target vehicle.
4. **Use scanning tools (e.g., IMSI-catcher or LTE-Tracker) to detect nearby vehicles' telematics connections.**
    - Deploy tools such as IMSI-catchers or LTE-Tracker to detect nearby connected vehicles by capturing their unique identifiers (IMSI, IMEI). These tools help you confirm if a vehicle's telematics ECU is in range and whether it's attempting to communicate with your rogue base station.
5. **Force a telematics ECU to connect via rogue broadcast beacons.**
    - Send out customized broadcast beacons that advertise your rogue tower as a higher-priority cell. Most ECUs are configured to connect to the strongest known network — so this trick compels the vehicle's telematics unit to drop its existing connection and establish one with your setup.
6. **Capture or manipulate OTA (Over-The-Air) SMS/FOTA updates sent from OEM servers.**
    - Once connected, monitor or intercept OTA messages, such as SMS-based commands or FOTA (Firmware-Over-The-Air) updates from the vehicle manufacturer. This enables you to capture legitimate update traffic or even spoof and inject your own messages as if they came from the OEM.
7. **Inject malicious payload, e.g., fake update binary or config redirect.**
    - Use the captured OTA session to deliver a malicious payload, such as a fake firmware binary, a redirect to a rogue update server, or a reconfigured telematics setting. This can let you alter system behavior, redirect data to attacker-controlled endpoints, or plant persistent backdoors.
8. **Trick the vehicle into downloading and "applying" unsigned firmware or ECU reconfiguration.**
    - Many vehicles lack proper firmware signing enforcement. By mimicking the format and metadata of legitimate updates, you can make the vehicle accept and apply an unsigned firmware image or configuration change, especially if the verification step is weak or skipped in its update logic.
9. **Log telemetry packets from vehicle to analyze what was accepted.**

- While the rogue station is active, capture outbound telemetry from the vehicle to study how it responds to injected updates or commands. This may include logs of successful firmware changes, error reports, or reboots—helping you understand what modifications were applied and whether persistence was achieved.

10. **Keep the station active while the vehicle is in proximity or moving slowly (e.g., at charging points or garages).**

    - Maintain your rogue base station's activity in areas where the vehicle remains for extended periods, like charging stations, service centers, or parking lots. This increases the chance of sustained interaction with the telematics unit, giving enough time to complete large data transfers or complex attacks undisturbed.

- **Procedure 2:** Cellular-Based Command Injection

1. **Spin up the rogue eNodeB and broadcast identical cell ID as OEM carrier.**

    - Set up a rogue cellular base station (eNodeB) that mimics the real OEM carrier by cloning its Cell ID and other identifiers. This tricks nearby vehicles into connecting to your fake tower, assuming it's a legitimate network.

2. **Identify IMSI and attach the vehicle's modem to your base station.**

    - Once a vehicle connects, extract the IMSI (International Mobile Subscriber Identity) from the initial handshake and lock the telematics unit to your rogue station, isolating its communications from the real network.

3. **Use silent SMS or carrier-specific commands to initiate interaction.**

    - Send silent SMS or vendor-specific signaling messages (which don't appear on the user interface) to probe the telematics unit and trigger backend interactions or diagnostic responses without alerting the user.

4. **Send a crafted SMS containing command to disable updates or open TCP ports.**

    - Craft and deliver a binary SMS that disables security updates or opens communication ports on the vehicle's system, setting up a future foothold for remote control or malware deployment.

5. **Intercept response packets (SMS/UDP/TCP) for confirmation.**

    - Monitor and capture return packets over SMS, UDP, or TCP to verify that the malicious command was executed successfully and determine the vehicle's current state or vulnerabilities.

6. **Deploy HTTP/S server that mimics OEM firmware update site.**

- Host a fake firmware update server that uses the same domain format, certificates, and file structure as the OEM's real server, to deceive the vehicle into connecting and requesting updates from the attacker.

7. **Respond with a malicious binary or altered version of config.dat.**
    - Serve a tampered firmware binary or configuration file to the vehicle containing backdoors, disabled checks, or logic bombs that give long-term control or sabotage key functionality.
8. **Modify the vehicle's logic (e.g., disable safety alerts, reflash modules).**
    - Reprogram ECUs or change parameters to disable critical alerts (e.g., brake warnings, proximity sensors), or completely reflash modules with attacker-controlled firmware to affect behavior.
9. **Initiate reboot or reset to apply changes.**
    - Send a command to restart the telematics system or vehicle ECUs, ensuring the newly applied malicious settings or firmware take effect immediately or on the next startup.
10. **Monitor C2 traffic and device logs over LTE or fallback 3G band.**
    - Use the fake base station to monitor command-and-control traffic and system logs, confirming successful compromise and maintaining visibility or access as the vehicle moves in and out of range.

📡 **Technique #2: ATE-004.02 – GNSS Jamming**

🔹 **Procedure 1:** Localized GNSS Signal Disruption

1. **Acquire a GPS/GLONASS/Galileo jammer (commercially available or DIY SDR-based).**
    - This step involves obtaining a jamming device that can disrupt satellite signals, either by purchasing a prebuilt model from grey-market sources or building one using software-defined radio platforms like HackRF with GNSS-spoofing scripts. DIY builds offer custom frequency tuning, while commercial units may be plug-and-play.
2. **Position jammer near test vehicle (e.g., roadside, undercarriage).**
    - Placement is key for effective disruption. Jammer should be close enough to interfere with the GNSS antenna—either on the dashboard, roof, or windshield. Hiding it under the car or placing it on roadside poles ensures signal blockage without being visible.
3. **Start jamming during route traversal or ignition.**

- Turning on the jammer when the vehicle starts or during movement interrupts real-time satellite communication, causing immediate navigation faults. This helps test whether the vehicle has redundancy like inertial sensors or relies heavily on satellite fixes.

4. **Observe dashboard: check for loss of satellite fix and "No GPS" warnings.**
    - Once jamming starts, the infotainment or HUD may throw alerts like "Searching for GPS," "Location unavailable," or even navigation freezes. These are visual signs of signal loss and how the system prioritizes alerting the driver.
5. **Cross-check with tools like u-center or GNSS receivers via OBD.**
    - Diagnostic software like u-blox's u-center can be connected to the GNSS receiver to confirm absence of satellite data. Similarly, OBD-II interfaces with apps like Torque or OBDeleven help verify drop in satellite signal count and timestamp irregularities.
6. **Note fallback behavior—dead reckoning or sensor fusion without GPS.**
    - Vehicles may switch to alternate positioning modes using wheel sensors, accelerometers, or gyros (sensor fusion). Tracking how long the system can operate in fallback mode without major drift reveals resiliency.
7. **Monitor error codes (e.g., DTCs) via OBD-II scan tools.**
    - Fault codes generated during GNSS disruption can be extracted with scan tools. These may include location module failures, navigation system warnings, or time-sync mismatches affecting ECUs relying on GPS clock.
8. **Review camera overlays, map misalignments, or assist failures.**
    - GNSS loss can break map alignment with camera feed in AR overlays or HUDs, leading to incorrect road detection. Lane departure, navigation prompts, and assisted braking might malfunction if localization fails.
9. **Capture CAN traffic anomalies or broadcast retries from the GNSS module.**
    - Monitoring the CAN bus shows increased message retries from the GNSS module, or silence in data broadcast intervals. This disruption pattern gives insights into how the vehicle handles comms failure internally.
10. **Optionally modulate jamming signal to alternate GNSS bands to evade detection.**
    - Switching between L1, L2, and E1/E5 bands avoids persistent jamming detection by basic anti-jam logic. It simulates real-world interference and tests whether the vehicle switches GNSS constellations (e.g., GPS to Galileo) effectively.

- **Procedure 2:** Assisted Driving Failures via Jamming

1. **Choose vehicle with Level 2 or higher ADAS relying on GNSS.**
    - Select a modern vehicle equipped with Advanced Driver Assistance Systems (ADAS) such as lane-centering, adaptive cruise control, and auto-pilot functionalities. These systems often depend heavily on Global Navigation Satellite Systems (GNSS) for accurate positioning. Verifying that the vehicle uses GPS/GNSS as a critical input will ensure the effectiveness of your jamming test on assisted-driving behaviors.
2. **Place the jammer near roof antenna or rear deck.**
    - Install the GNSS jammer close to the car's GNSS receiver, typically found integrated into the shark-fin antenna on the roof or near the rear windshield. The goal is to reduce the received satellite signal strength directly at the source, minimizing the risk of detection while maximizing disruption effectiveness.
3. **Trigger jamming when ADAS systems are active (e.g., lane keeping, auto-cruise).**
    - Activate the jammer once the vehicle's ADAS systems are engaged. For example, during highway driving with lane-keeping and adaptive cruise control enabled. Disrupting GNSS during active navigation lets you directly observe system instability or emergency disengagement as it loses reliable positioning data.
4. **Record real-time driving anomalies (e.g., aggressive braking, system disengage).**
    - Closely monitor and document erratic vehicle behaviors such as sudden braking, ping-pong lane movements, steering handover requests, or full ADAS disengagement. Use an in-cabin camera and dashboard logging to capture these anomalies in real time for later analysis and reporting.
5. **Use diagnostic software to fetch logs post-incident.**
    - Connect OBD-II diagnostic tools or use the manufacturer’s debug console to extract vehicle logs after the jamming session. Focus on timestamps where anomalies occurred and check for error codes, GNSS-related faults, or subsystem crash logs to correlate with physical behavior observed.
6. **Note impact on OTA behavior (some updates require location trust).**
    - Certain OTA (Over-the-Air) update mechanisms use location integrity to verify eligibility for installing critical firmware patches. Observe whether GNSS jamming impacts the vehicle’s ability to check for or initiate such updates, potentially delaying essential system patches.
7. **Force vehicle to enter degraded mode or alert “Navigation not available.”**
    - Sustained jamming typically forces the vehicle to enter a fallback or degraded driving mode, disabling advanced features and displaying warning messages

like “Navigation Unavailable” or “ADAS Limited.” These warnings are critical indicators of GNSS dependency within the ADAS stack.

8. **If jammed long enough, inspect if the vehicle suspends further OTA.**
    - Extended denial of GNSS data may lead the system to postpone or suspend firmware updates indefinitely, interpreting the lack of trusted location data as a security or reliability risk. This behavior highlights how adversaries could halt critical update cycles by disrupting satellite input.
9. **Trigger edge case behaviors (e.g., under tunnels + jamming).**
    - Combine physical coverage areas like tunnels or parking garages with active jamming to create compound signal loss scenarios. These edge cases often stress-test the fallback logic and may reveal weaknesses in how ADAS or OTA systems recover from combined disruptions.
10. **Document impact on user trust and system fallback safety.**
    - Finally, evaluate the broader implications of GNSS jamming on user confidence and vehicle fail-safes. A system that disengages with unclear alerts or dangerous behavior may undermine trust in automation. Recording these effects contributes to understanding real-world safety concerns during signal denial attacks.

🔨 **Technique #3: ATE-006.03 – Blinding the Camera**

🔹 **Procedure 1:** Optical Interference on ADAS

1. **Identify the camera module (e.g., forward-facing lane camera).**
    - Begin by locating the key camera responsible for advanced driver assistance systems (ADAS), such as the one mounted near the rearview mirror or on the windshield. These modules are typically used for lane detection, traffic sign recognition, and forward collision alerts. Knowing the exact module helps you target the right sensor for testing.
2. **Use 3000+ lumen flashlight or laser pointer for interference.**
    - Use a high-intensity light source—either a commercial-grade flashlight exceeding 3000 lumens or a focused laser pointer. These devices can temporarily blind optical sensors. The light intensity overwhelms the camera’s sensor, simulating a scenario where the vehicle cannot “see” ahead due to extreme brightness or glare.
3. **Position yourself at distance typical for oncoming headlights.**
    - Set the interference source at a distance that mimics real-world headlight angles, roughly 15–30 meters away. This ensures the test environment mirrors

natural conditions, like approaching vehicles at night. It's crucial to keep angles and height in sync with how the sensor would encounter headlights during night driving.

4. **Match the LED strobe to the rolling shutter rate of the CMOS sensor.**
    - Use a frequency-adjustable LED strobe light and synchronize it with the CMOS camera's rolling shutter speed (usually between 30–120 Hz). Matching the strobe pulse to the sensor's refresh rate causes visual artifacts—like banding or flicker—ultimately impairing the camera's ability to capture clean frames for ADAS processing.
5. **Observe whiteout/flicker in camera feed using vehicle service tools.**
    - Access the live video feed from the affected camera using OEM diagnostics tools or a dealer-grade service console. Look for visual artifacts like complete whiteouts, blurring, or rapid flicker. These visual glitches signal that the camera is struggling to interpret its environment accurately.
6. **Simulate nighttime driving scenario for maximum effect.**
    - Conduct this test in low-light or nighttime conditions to amplify the impact. Cameras switch to high-gain sensitivity modes at night, making them more susceptible to intense light. This condition increases the likelihood of ADAS failure or misjudgment when exposed to interference.
7. **Trigger lane keep assist, adaptive cruise, or braking—watch for failures.**
    - While the interference is active, engage driving features that rely on visual input. Systems like lane keeping or adaptive cruise control may become erratic, deactivate suddenly, or respond incorrectly (e.g., phantom braking). Monitor and log each behavior as the vehicle attempts to operate with a blinded camera.
8. **Record error messages like "Camera temporarily blocked."**
    - Modern vehicles often generate warnings when a sensor becomes inoperative. Expect alerts such as "Camera view obstructed" or "ADAS unavailable." Document these messages as part of the test log—they serve as visible indicators of the vehicle's internal failure detection mechanisms.
9. **Verify fallback: does it alert the driver or switch to manual?**
    - Assess how the system responds to persistent blinding. Ideally, the vehicle should inform the driver and disable ADAS features, forcing manual control. Confirm whether these fallback behaviors are triggered properly and if they appear consistently across different vehicle models.
10. **Repeat with angled light at side cameras and interior IR driver monitor.**

- Extend the test by targeting secondary vision systems like side cameras (used in blind-spot monitoring) and the interior infrared (IR) camera used for driver attention monitoring. Evaluate how different angles, light types, or reflection patterns affect these sensors' ability to function correctly under intentional light interference.

- **Procedure 2:** Saturating Surround Camera Suite

1. **Target surround view or bird's-eye cameras during parking or reverse.**
    - Begin by identifying when the vehicle activates its 360-degree or rearview camera systems—usually during reverse gear or slow-speed parking. These systems rely on multiple exterior cameras (often front, rear, and side-mounted), which feed data to a central ECU for real-time stitching and display. This makes them ideal for manipulation in low-movement scenarios when drivers depend on visuals over mirrors.
2. **Shine diffused IR or white LED from multiple directions (to simulate sun flare).**
    - Use diffused high-intensity IR or white LED lighting aimed from angles that mimic natural glare, such as bright sunlight at dawn or dusk. Aim to confuse auto-exposure, white balance, and HDR logic within the camera sensors. This can lead to washed-out or overly darkened visuals, simulating real-world lighting challenges and forcing the system into unreliable states.
3. **Record poor rendering or no-display condition in central HMI screen.**
    - Monitor the vehicle's Human-Machine Interface (HMI), where stitched camera views are displayed during maneuvering. You may observe glitches like ghost images, missing feeds, or entire view blackouts. Record and document these events using either onboard screen capture (if accessible) or external video recording for later analysis of the attack's effectiveness.
4. **Test auto-park or reverse-assist—it may abort or trigger false object detection.**
    - Once video feed degradation is confirmed, activate auto-park or reverse-assist features. These systems use both visual and sensor data to operate. Manipulated visuals may cause the system to misjudge distances, falsely detect obstacles, or completely fail to initiate parking maneuvers. This not only undermines driver trust but also puts vehicles at risk of collisions.
5. **Log events from onboard diagnostics via OBD or Ethernet debug port.**
    - Connect to the vehicle's OBD-II port or, if accessible, its Ethernet debug interface to pull diagnostic logs. Look for fault codes related to camera modules (often labeled as CAM_FL, CAM_RR, etc.) or ECU errors indicating loss of video feed, overexposure, or timeout events. This technical evidence is crucial for validating the disruption beyond visible symptoms.
6. **Adjust light patterns to test for sensor fusion confusion.**

- Change the angle, pulsing rate, or intensity of your lighting to see how the system adapts. Since many vehicles rely on sensor fusion—combining video, radar, and ultrasonic input—observe whether it fails to reconcile contradictory data (e.g., radar detects an object, but camera is blind). This can lead to unstable or failed decisions during maneuvering.

7. **Attempt to blind multiple cameras simultaneously for greater disruption.**
    - Scale the attack by synchronizing lights to target more than one camera at a time. This increases the likelihood of full system collapse since modern vehicles often require a minimum number of operational cameras for 360-degree stitching. Multi-angle blinding raises chances of inducing total feature deactivation (e.g., "Surround view unavailable").
8. **If using a known camera vendor (e.g., Mobileye), use frequency-specific attacks.**
    - If the target vehicle is known to use specific camera vendors like Mobileye or Continental, research their sensor specs (e.g., rolling shutter speed, anti-flicker algorithms). Tailor your lighting frequencies or strobe rates to exploit known limitations in those systems, increasing the probability of sensor disruption without triggering alerts.
9. **Review CAN logs for camera timeout or image processing error codes.**
    - Access the vehicle's Controller Area Network (CAN) logs to look for specific DTCs (Diagnostic Trouble Codes) or timeout flags. These entries can reveal if image streams were lost, if ECUs failed to receive camera data within expected intervals, or if video frames were corrupted. Such logs often go unnoticed by drivers but are key for back-end diagnostics.
10. **Confirm persistent disruption (e.g., after light is removed, reboot required?).**
    - After removing the light source, observe whether the system recovers instantly or requires a vehicle reboot to restore normal camera function. Persistent issues suggest deeper disruption—possibly triggering fail-safe modes or requiring manual service reset. This level of failure can help demonstrate the severity and potential real-world impact of the attack.

🔁 **Attack Flow Summary**

- **Step 1**: Hijack cellular trust chain with rogue base station.
- **Step 2**: Deny GNSS signals to break position-based safety systems.
- **Step 3**: Blind vision stack to fail perception-dependent automation.

⚔️ #2 - Tactic: Initial Access (ATA-002)

Objective: Gain unauthorized entry into the vehicle network or system, typically by exploiting physical, wireless, or remote entry points.

🔧 Technique #1: Exploit Physical Diagnostic Interface (ATA002-T001)

▶️ **Procedure 1:** Unauthorized OBD-II Port Access via External Scan Tool

1. Locate the OBD-II port beneath the steering wheel or near the driver's footwell — standard in most modern vehicles.
2. Bring a commercially available ELM327 Bluetooth scanner or a more advanced device like the Bosch KTS or Autel MaxiSys.
3. Connect the device discreetly when the vehicle is parked and unlocked or during servicing in a garage environment.
4. Power up the scanner and initiate a handshake with the car's ECU (typically over CAN protocol).
5. Use the scanner software to list available modules — look for TCU (telematics), ABS, ADAS, or BCM (body control module).
6. Attempt unrestricted read/write commands using UDS or KWP2000 protocol supported by many OEMs.
7. Download DTCs (Diagnostic Trouble Codes), VIN, firmware versions, and potentially encryption keys if stored plaintext.
8. Try to unlock ECU write access — some modules support session control requests or seed-key algorithms.
9. Send crafted payloads (e.g., fake firmware version) to simulate a corrupted or outdated module needing an OTA update.
10. Disconnect the scanner; data exfiltration is complete. If payload was injected, it may execute after ignition cycle or OTA ping.

▶️ **Procedure 2:** Malicious USB Injection via Service Mode Port

1. Identify the vehicle's USB port (used for infotainment updates or diagnostics), often in the glove box or center console.
2. Prepare a USB drive mimicking OEM structure — folder names like /update, /firmware, or /navdata based on reverse engineering.

3. Include a malformed or trojanized config.dat or update.img with embedded shellcode or buffer overflow triggers.
4. Rename file extensions to expected types (e.g., .bin, .img, .pak) and ensure timestamp spoofing for freshness.
5. Insert USB stick while vehicle is in Accessory Mode or Maintenance Mode (activated via key cycle or secret menu).
6. If infotainment system autoscans USB, it may automatically open the update prompt or silently parse the content.
7. Exploit weak firmware signature verification — common in Tier-1 infotainment vendors like Harman or Continental.
8. Payload executes in sandboxed Linux shell or QNX environment, gaining access to /dev partitions or exposed APIs.
9. Inject persistent scripts (e.g., cronjob) or set a trigger for future ignition cycle execution.
10. Remove USB after injection; if successful, the malware stays dormant until activated or exfiltrates data via CAN.

🛰️ Technique #2: Exploit Cellular Telematics Interface (ATA002-T002)

▶️ Procedure 1: IMSI Catcher with Rogue eNodeB to Hijack Telematics

1. Deploy a rogue LTE base station using software-defined radio (SDR) tools like srsRAN or OpenLTE on a USRP or BladeRF.
2. Configure the rogue eNodeB with same MCC-MNC as local carrier (e.g., 404/45 for India) and high signal power to force handover.
3. Broadcast identical Cell ID and track the target vehicle's IMSI using silent paging or identity requests.
4. When vehicle connects, initiate attach procedure to capture NAS messages and negotiate basic communication.
5. Use silent SMS to probe application-layer responses or trigger modem debug channels.
6. Once command channel is open, push custom binary SMS or control message that opens TCP port or disables OTA validation.
7. Log inbound TCP beacon packets or confirmation SMS indicating command received.

8. Simulate OTA update flow by hosting malicious HTTP/S server mimicking OEM domain structure.
9. Redirect update checks using DNS poisoning or modified APN config, forcing vehicle to download fake firmware.
10. After payload is delivered and executed, monitor persistent callbacks to rogue C2 server via fallback 3G or LTE bands.

▶️ **Procedure 2:** Brute-force API Tokens from OEM App Using Known VINs

1. Scrape public vehicle sales databases or VIN decoders to collect real VIN numbers from the same OEM.
2. Reverse engineer the official OEM mobile app (APK decompilation) to locate API endpoints (e.g., /auth, /vehicle/{vin}).
3. Identify static secrets, bearer tokens, or predictable token generation tied to VIN+timestamp or VIN+IMEI.
4. Use a script to attempt brute-force or token-replay attacks using previously seen VIN-auth pairs.
5. If response includes vehicle status (locked/unlocked, GPS), confirm live telematics access.
6. Send crafted commands like lock=false, honk=true, or initiate OTA check using captured session token.
7. Exploit weak rate limiting or misconfigured access control that doesn't validate session integrity.
8. Attempt firmware downgrade or trigger system crash via malformed update metadata.
9. Reflash custom software modules using the API if developer/debug endpoints are exposed.
10. Maintain session persistence by capturing refresh tokens or rotating device fingerprints.

🌐 **Technique #3: Exploit Wireless Key Fob Interface (ATA002-T003)**

▶️ **Procedure 1:** Relay Attack with Two SDR Devices on Passive Entry System

1. Identify vehicles supporting passive keyless entry (PKE) — typically within 5 meters of the fob.

2. Place one SDR device (transmitter) near the owner's house or pocket and another (receiver) near the vehicle.
3. SDR captures low-frequency (LF) signal from car, mimicking a proximity check.
4. Transmit LF signal to the other SDR via Wi-Fi or dedicated radio bridge in real-time (low latency required).
5. The second SDR rebroadcasts LF to fob, triggering it to respond with unlock/authorize token.
6. Response token is captured by receiver and sent back to the transmitter SDR.
7. Car receives valid response, thinking the key is nearby, and unlocks or powers on.
8. Attacker can now drive away, assuming the vehicle doesn't use motion-detection or fob-presence rechecks.
9. If motion sensors are active, spoof accelerometer data from a dummy fob or inject false presence via CAN.
10. Log timestamps and attempt this with multiple vehicles to check resistance against rolling code and PASE timeout.

▶ **Procedure 2:** Brute-force Rolling Code via RF Replay Attack

1. Capture RF signals from key fob when user locks/unlocks the car — usually 315/433/868 MHz bands.
2. Use a HackRF or Yard Stick One to record several signal bursts, then demodulate using GNU Radio.
3. Identify preamble, sync word, payload, and CRC using tools like URH (Universal Radio Hacker).
4. Analyze rolling code algorithm — many use simple LFSR or counter-based schemes with known weaknesses.
5. Generate a set of plausible next codes by incrementing sequence or bruteforcing small keyspace.
6. Replay these codes toward the vehicle in successive attempts, with millisecond-level timing precision.
7. If accepted, car unlocks or performs function (like trunk open), confirming successful bypass.
8. Use same method to intercept keyless start packets or fob pairing signals.
9. Attempt denial-of-service by exhausting the key's allowable sync range (code desync).

10. Record RF logs and test defense mechanisms — such as key invalidation after N failed attempts.

🚙 PoC #3: Execution (ATA-003) – MITRE Automotive Threat Matrix

---

Technique 1: ATA0031 – Malicious Firmware Execution

Procedure 1A: Flash Custom ECU Firmware via UDS Reprogramming

1. Obtain access to the car's OBD-II port using a physical connector and ensure the car is in diagnostic or maintenance mode.
2. Use tools like UDS-Python, OpenXC, or BusMaster to initialize communication with the ECU over CAN or UDS protocol.
3. Identify the ECU's session access control level—typically, it needs to be elevated using a Security Access request (seed-key challenge).
4. Extract the security seed, and calculate or brute-force the key using known algorithms for that specific vehicle make.
5. Once access is granted, issue the command to enter the reprogramming mode (e.g., UDS 0x10 service).
6. Prepare a malicious firmware image tailored for the target ECU—this might disable or subvert key functions like speed limiter or traction control.
7. Use UDS service 0x36 (Request Download) and 0x37 (Transfer Data) to stream your malicious firmware into the ECU.
8. Conclude the flashing process using 0x38 (Request Transfer Exit) and a reset command to reboot the ECU into normal mode.
9. Monitor for successful execution of malicious logic, e.g., changing vehicle parameters or disabling alert messages.
10. Validate that the ECU boots with the new firmware and remains undetected by standard diagnostics or dashboard alerts.

Procedure 1B: Modify Existing Firmware Binary, Then Flash via Bootloader

1. Dump the existing firmware from the ECU using a hardware debugger like JTAG, or extract via diagnostic protocols if allowed.
2. Analyze the firmware image in IDA Pro or Ghidra, focusing on entry points and CAN-handling logic.

3. Identify logic responsible for safety-critical functions like braking or steering assistance.
4. Inject custom shellcode or NOP-out critical safety checks to induce failure or create backdoors.
5. Patch CRC and firmware integrity checks, often hardcoded or enforced during boot validation.
6. Repack the modified firmware image using appropriate formats (e.g., Motorola S-record, Intel HEX).
7. Enter the ECU's bootloader mode using GPIO or pin shorting techniques if not accessible via OBD.
8. Upload the malicious firmware using flashing tools compatible with the specific microcontroller (e.g., ST-Link, Bosch UDS flasher).
9. Power cycle the ECU and observe if your firmware persists through cold starts.
10. Verify your code executes in live driving conditions, ensuring it alters behavior without triggering dashboard alerts.

Technique 2: ATA0034 – Execution via Infotainment APIs

Procedure 2A: Abuse Media Parsing Vulnerability via USB

1. Craft a malformed media file (e.g., MP4, FLAC) that exploits a known parsing vulnerability in the infotainment system's codec libraries.
2. Load the malicious file onto a USB stick with a typical structure mimicking user behavior.
3. Insert the USB into the infotainment port while the vehicle is running or in accessory mode.
4. Let the media indexer or file scanner of the system automatically process the file.
5. Upon parsing, the exploit triggers a buffer overflow or command injection vulnerability in the infotainment OS.
6. Use the exploit payload to gain shell access or arbitrary code execution privileges.
7. Establish persistence by dropping scripts in startup directories or modifying system services.
8. From this foothold, laterally access other CAN-connected modules if the infotainment unit is bridged with driving systems.
9. Exfiltrate data like paired phones, contact logs, or GPS history from the infotainment memory.

10. Test across multiple vehicle models with similar infotainment firmware to assess exploit reliability.

Procedure 2B: Exploit Web-Based Infotainment Browser Over Wi-Fi

1. Identify vehicles with infotainment systems that offer web browsing or OTA updates over in-car Wi-Fi.
2. Set up a rogue access point mimicking a known trusted SSID (e.g., "Toyota_Free_WiFi") using hostapd.
3. Lure the infotainment unit to auto-connect by spoofing MAC address and providing internet access.
4. Host a malicious webpage containing JavaScript or iframe exploits targeting known infotainment browser flaws.
5. Wait for the user to interact with the page or let auto-refresh features render it silently.
6. Exploit browser sandbox escapes or system call flaws to run arbitrary commands.
7. Drop a script that runs on boot (e.g., modifies navigation system, shows fake alerts).
8. Use lateral network scanning from the infotainment system to discover connected vehicle ECUs.
9. Inject CAN messages or use known APIs if the infotainment stack allows bridge access to control modules.
10. Validate persistence and stealth by rebooting the unit and checking if the payload survives.

Technique 3: ATA0037 – Remote Command Injection

Procedure 3A: Exploit Command Injection via Telematics Update Server

1. Identify the telematics system vendor and its firmware update server endpoint (e.g., update.telematics.vendor.com).
2. Use traffic analysis (Wireshark or mitmproxy) to intercept update calls during a legitimate OTA event.
3. Modify the response to include shell metacharacters or payloads in headers or filenames.
4. Inject commands into firmware metadata fields like description, which may be parsed by shell scripts on the IVI unit.
5. Host a spoofed update server delivering the crafted response and DNS-spoof the domain.

6. Upon vehicle check-in, it fetches the malicious metadata and executes commands if not sanitized.
7. Use payloads like reverse shells, system recon, or log tampering commands in the injection string.
8. Observe if the telematics control unit logs or reboots unexpectedly—an indicator of successful code injection.
9. Set up callbacks to your C2 server to capture shell or execution proof.
10. Test whether reboots or network disconnections disable the malicious execution, helping define persistence limits.

Procedure 3B: Exploit Cloud APIs with Weak Command Sanitization

1. Research exposed cloud APIs used by vehicle apps (e.g., remote lock/unlock, locate vehicle).
2. Use fuzzing tools to inject payloads into command fields (e.g., name=";reboot").
3. Identify improper input handling on the backend (e.g., Python eval, unsanitized shell calls).
4. Craft a POST request to invoke the API with a malicious payload targeting execution on vehicle-bound scripts.
5. Observe the vehicle behavior using a paired app or physical presence to confirm any unusual actions.
6. Verify command injection by attempting benign payloads (e.g., sleep 5) and measure the delay.
7. If successful, escalate to destructive commands like clearing logs or disabling security functions.
8. Log the server responses and exploit logs to document execution results.
9. Use network logs to identify source servers where execution occurred (e.g., telematics backend vs vehicle itself).
10. Confirm if the exploit can be repeatedly executed or patched by OTA update.

#4 - **Tactic: Persistence (ATA-004)**

Objective:
The goal of the Execution tactic is to **run malicious code or commands** inside the vehicle's systems after gaining access. This enables the attacker to **manipulate behavior**, **bypass safety**, or **trigger unsafe operations** by activating scripts, exploiting APIs, or injecting code into ECUs or infotainment units. It's the step where an attack goes from passive access to active disruption.

Technique 1: Modify Bootloader Configuration

Procedure 1: Inject Persistent Malware via U-Boot Script Modification

1. **Access ECU firmware image**: Extract the ECU firmware using manufacturer tools or chip-off techniques from flash memory.
2. **Mount filesystem**: Use tools like Binwalk or QEMU to mount the firmware image and locate the bootloader (often U-Boot).
3. **Identify boot sequence**: Find the U-Boot environment script (e.g., bootcmd, bootargs) that launches the OS kernel or init scripts.
4. **Modify script**: Insert a line to launch a malicious binary (e.g., /mnt/data/malware.bin) on each boot.
5. **Ensure file system persistence**: Place the binary in a writable and non-volatile partition such as /mnt/data/ or /persist/.
6. **Repack image**: Use firmware tools to repack the modified image and calculate checksums if required.
7. **Reflash the ECU**: Flash the modified firmware back to the vehicle using OBD reprogramming tools or via JTAG/UART.
8. **Verify execution**: After reboot, check if the malware is active and retains state across multiple reboots.
9. **Test stealth**: Ensure the malware hides itself from normal diagnostics or OTA checks.
10. **Monitor logs**: Review logs and CAN traffic to confirm malware executes consistently after each boot cycle.

Procedure 2: Modify Secure Boot Validation Routine

1. **Gain access to secure bootloader binary**: Dump the bootloader code (via hardware debug or firmware extraction).
2. **Disassemble bootloader**: Use tools like Ghidra or IDA Pro to analyze the validation routine (e.g., RSA signature check).
3. **Locate verification logic**: Identify code that verifies kernel or initrd signatures before loading.
4. **Patch logic**: Modify or bypass the validation code so that unsigned code is accepted silently.
5. **Insert backdoor loader**: Add code that loads a secondary stage bootloader or malware image conditionally.
6. **Repack and realign**: Ensure that the modified binary meets all offset/alignment and checksum expectations.
7. **Flash modified bootloader**: Write it back to the device using UART/JTAG or OTA if the system permits rollback.
8. **Test boot chain**: Confirm that the vehicle boots into the OS with the patched bootloader.
9. **Run unsigned code**: Deploy unverified applications or binaries post-boot without triggering integrity errors.
10. **Persist changes**: Ensure that manufacturer updates do not overwrite the patched loader.

Technique 2: Install Malicious Background Daemon

Procedure 1: Add Init.d or SystemD Service for Malware Execution

1. **Identify init system**: Determine whether the vehicle uses init.d, SystemD, or custom startup manager.
2. **Create startup script**: Write a shell script or service unit that launches your payload.
3. **Drop payload in writable partition**: Store malware in /usr/bin/, /mnt/data/, or /opt/ depending on write permissions.
4. **Set correct permissions**: Ensure execution (chmod +x) and proper ownership so it runs with required privileges.
5. **Enable on boot**: For SystemD, run systemctl enable malware.service or add script to rc.local for init.d.
6. **Test manually**: Run script by hand and verify process starts with proper arguments.

7. **Reboot the system**: Observe if malware runs automatically and remains persistent post-reboot.
8. **Hide process**: Rename process or move to background with misleading name like system-update.
9. **Monitor health**: Use system tools to verify daemon remains active (e.g., respawn via watchdog).
10. **Log activities**: Store logs in hidden or obscure location to monitor malware health without detection.

Procedure 2: Use Cron or Timed Jobs to Trigger Actions

1. **Access crontab or scheduler**: Determine whether the OS uses cron, atd, or custom job scheduler.
2. **Create periodic job**: Insert a line in /etc/crontab or user-specific crontab to run malware every X minutes.
3. **Choose effective interval**: Schedule actions during vehicle idle periods to avoid detection (e.g., midnight).
4. **Add obfuscation**: Encode the payload call using base64 or indirect script references.
5. **Set permissions**: Assign appropriate user/group permissions to avoid alerts from privilege mismatch.
6. **Test the task**: Wait for scheduled time and verify execution in process logs or malware logs.
7. **Ensure persistence**: Reboot system and validate cron jobs remain intact.
8. **Monitor execution**: Capture STDOUT/STDERR to a hidden log file to troubleshoot or monitor success.
9. **Bypass cleanup**: Prevent OEM scripts from wiping temp or cron directories by moving tasks elsewhere.
10. **Adjust schedule remotely**: If system has backdoor, allow cron interval to be modified via command and control (C2).

Technique 3: Exploit Diagnostic Persistence Settings

Procedure 1: Use UDS WriteDataByIdentifier for Config Injection

1. **Connect via OBD-II**: Use a UDS-capable tool (e.g., CANalyzer, Python UDS library) to access ECUs.

2. **Authenticate session**: Switch to extended diagnostic session using 0x10 service.
3. **Send security access challenge**: Complete 0x27 key exchange to unlock write permissions.
4. **Use 0x2E service**: Issue a WriteDataByIdentifier command to alter persistent parameters.
5. **Modify startup scripts**: Inject path to custom binary in diagnostic config (if supported).
6. **Send 0x31 request**: Trigger routine control to reload config files.
7. **Reboot ECU**: Recycle power or use UDS reset service (0x11) to reload modified config.
8. **Confirm persistence**: Reconnect post-reboot and verify altered behavior or malware presence.
9. **Check DTCs**: Ensure no diagnostic trouble codes were generated by your config injection.
10. **Repeat for other ECUs**: Target infotainment, telematics, or ADAS modules if vulnerable.

Procedure 2: Abuse OEM Debug Mode for Persistence

1. **Identify hidden debug trigger**: Use service tools or leaked docs to discover OEM-specific debug activation.
2. **Enable debug mode**: This may be via button combos, UDS messages, or hidden USB config flags.
3. **Gain shell access**: In debug mode, ECUs may expose a Linux shell or developer backdoor.
4. **Remount filesystem RW**: Temporarily remount / or /etc/ as read-write.
5. **Inject startup scripts**: Insert payload into init.rc or similar startup configuration.
6. **Lock-in debug mode**: Modify system flag to persist debug even across resets or OTA.
7. **Hide changes**: Use timestamps, process obfuscation, and logs cleanup to evade detection.
8. **Test recovery**: Confirm if factory reset or safe mode removes persistence—if not, the attack is effective.
9. **Remove breadcrumbs**: Wipe shell history, UDS session logs, and metadata.
10. **Document fallback**: Note any alternate interfaces (USB, Ethernet) left open as future access points.

⚠️ Tactic: Privilege Escalation (ATA-005)

Goal: Gain higher-level access on vehicle subsystems or ECUs to perform actions beyond the attacker's current permissions, such as reflashing firmware, disabling security features, or persisting within high-integrity domains.

✅ Technique 1: Exploit Debug Interface on Telematics ECU

ID: ATA-005-T001

📍 **Procedure 1:** Leveraging UART Interface to Gain Root Shell

1. Locate the telematics ECU behind the dashboard using service manuals.
2. Open the enclosure to expose the PCB and inspect for UART test points.
3. Connect USB-to-UART converter (3.3V) and monitor boot logs via serial console.
4. Interrupt the bootloader (e.g., U-Boot) by sending keystrokes on boot.
5. Dump memory or environment variables for bootargs or root access hints.
6. Modify kernel arguments to boot into single-user mode.
7. Restart the ECU and observe shell prompt without password enforcement.
8. Mount root filesystem with read-write permissions.
9. Modify /etc/passwd or install SSH server with custom credentials.
10. Maintain access by soldering pin header and sealing casing cleanly.

📍 **Procedure 2:** Abuse Exposed JTAG to Bypass Authentication

1. Scan exposed pins on the telematics board using a JTAGulator or multimeter.
2. Identify TCK, TMS, TDI, TDO, and ground through signal behavior analysis.
3. Connect with OpenOCD or similar tool to dump firmware.
4. Analyze the firmware offline using Ghidra for authentication routines.
5. Patch out the login check or insert backdoor credentials.
6. Flash the modified firmware back via JTAG.
7. Reboot and observe modified behavior (e.g., no password login).
8. Install privilege escalation tools or remote access scripts.
9. Test command execution as root without triggering alerts.
10. Maintain access via JTAG trigger or hidden debug service.

✅ Technique 2: Exploit Vulnerable System Service (Linux/Android-based IVI)

ID: ATA-005-T002

📍 **Procedure 1:** Local Privilege Escalation via SetUID Binary Exploit

1. Gain shell access via exposed USB or Wi-Fi debug services.
2. Enumerate binaries with find / -perm -4000 -type f 2>/dev/null.
3. Discover a misconfigured SetUID binary (e.g., custom update tool).
4. Reverse-engineer binary to identify insecure system call execution.
5. Craft payload that tricks the binary into executing a shell.
6. Execute the payload and verify elevated privileges using id.
7. Patch /etc/sudoers to allow passwordless root for persistence.
8. Disable audit logs or log rotation to erase tracks.
9. Set up a cronjob or systemd timer for reverse shell on boot.
10. Monitor system behavior to ensure elevation is stable.

📍 **Procedure 2:** Exploiting Binder Service in Android-based Infotainment

1. Identify vehicle running Android IVI system through documentation or UI.
2. Connect via ADB over USB or Ethernet.
3. Use service list to enumerate exposed Android Binder services.
4. Reverse vulnerable system app (e.g., media, nav) using jadx or apktool.
5. Identify AIDL function lacking permission enforcement.
6. Use service call or write a custom APK to call the function.
7. Trigger unintended execution (e.g., access to shell, mount root).
8. Install rootkit or su binary in system partition.
9. Remove logs from /data/system/dropbox and logcat.
10. Maintain root through boot script or modified system app.

✅ Technique 3: Bypass Secure Boot via ECU Downgrade

ID: ATA-005-T003

📍 **Procedure 1:** Reflash Older Signed Firmware Using OTA Mechanism

1. Capture OTA update process over cellular/LTE using Wireshark or proxy.

2. Extract firmware image and verify it's signed with OEM key.
3. Search archives or leaked sources for older, vulnerable signed images.
4. Construct spoofed OTA server or MITM proxy.
5. Push the older image to the target ECU via authorized update path.
6. ECU accepts the image due to valid signature despite older version.
7. Exploit known vulnerability (e.g., command injection in shell script).
8. Patch startup script to spawn root shell over serial.
9. Re-enable the latest OTA after escalation for stealth.
10. Log changes and extract root access tools before reboot.

📍 **Procedure 2:** Physical Reflash via Boot Pin & Bypass Secure Boot Checks

1. Disassemble ECU and locate boot mode pins using datasheets.
2. Hold boot pin high while powering on ECU to enter DFU/Flash mode.
3. Connect ST-Link, CAN flasher, or UART-based tool as per hardware.
4. Dump current firmware and verify signing/encryption scheme.
5. Modify or replace bootloader with unsigned version.
6. Flash unsigned or tampered image using physical interface.
7. Reboot ECU and confirm no Secure Boot error.
8. Inject payload to grant persistent root shell via init script.
9. Monitor for watchdogs or self-check failures.
10. Seal ECU and test from vehicle bus if tampering is undetected.

Tactic #6: Defense Evasion (ATA-006) — MITRE Automotive Threat Matrix

Goal: Bypass, disable, or avoid detection from onboard diagnostic systems, telematics monitoring, and OEM intrusion detection systems (IDS) within the vehicle.

⚙️ Technique 1: Firmware Obfuscation

Procedure 1: Modify ECU Firmware Checksums

1. Gain access to the ECU firmware either by dumping it via OBD-II/UART or sourcing leaked firmware versions online.
2. Use reverse engineering tools like Ghidra or IDA Pro to identify integrity verification routines in the binary, typically CRC32 or SHA-based checks.
3. Patch the checksum validation logic to always return a valid result, even if the firmware is altered.
4. Test the patched binary on a virtual ECU emulator (if available) or an identical physical module to verify execution.
5. Reflash the altered firmware into the ECU using diagnostic tools like UDS/DoIP protocols or custom CAN-based scripts.
6. Observe if the onboard diagnostic system or telematics flags the change—most stock systems won't catch silent patches.
7. Continue monitoring CAN bus for unusual alerts or flags set by IDS modules.
8. If alerts occur, add dummy watchdog routines to mimic expected behavior.
9. Re-run diagnostics tools (OEM or aftermarket) to verify that the changes are invisible to standard validation processes.
10. Maintain version control of firmware patches for rollback in case of system lockout or OTA overwrite.

Procedure 2: Encrypt Malicious Payloads in Flash Updates

1. Start with obtaining the encryption mechanism used in OTA firmware updates (often AES-GCM or RSA-wrapped AES).
2. Extract the vehicle's firmware update key using chip-off techniques or by intercepting the bootloader during a debug session.
3. Create a modified firmware payload that contains malicious logic (e.g., CAN injection routines) inside unused partitions.
4. Encrypt this binary using the OEM's encryption standard and rewrap it using the correct signing scheme.

5. Host a mock OTA update server to deliver this encrypted package when the vehicle attempts an update.
6. Redirect the vehicle's update check using DNS spoofing or APN manipulation through a rogue LTE base station.
7. Confirm that the car accepts the malicious update and decrypts it as if it were official.
8. Monitor execution of your payload post-installation via CAN behavior or logging routines.
9. Use obfuscation techniques like polymorphic code or encrypted config blocks to hide functionality from binary diffing.
10. Ensure OTA rollback is disabled to prevent your payload from being overwritten.

🧰 Technique 2: Log Manipulation

Procedure 1: Modify Diagnostic Trouble Codes (DTCs)

1. After performing unauthorized actions (e.g., CAN flooding or spoofing), DTCs will often be logged in one or more ECUs.
2. Connect to the vehicle using diagnostic tools like OBD-II scanners or UDS-based tools.
3. Enumerate stored DTCs using service 0x19 and retrieve DTC status bytes.
4. Use UDS service 0x14 (Clear Diagnostic Information) to erase specific DTCs or use 0x2E to overwrite memory values directly.
5. On some systems, DTCs are mirrored in shadow RAM—ensure both primary and secondary copies are wiped.
6. Patch firmware to disable DTC storage or change DTC severity from critical to informational.
7. Validate that DTC clearing does not trigger backup logs or fault snapshots.
8. Simulate driving behavior to confirm that the cleared codes do not reappear.
9. Modify the metadata used by logging modules to avoid crash dumps being written.
10. Monitor telematics reports, ensuring that OEM cloud platforms no longer receive historical fault traces.

Procedure 2: Spoof Timestamped Log Entries

1. Intercept the log write function inside the ECU or IVI system using debugger access or firmware patching.
2. Modify the logging routine to write fake timestamps and message contents.

3. Insert plausible entries such as “System running normally” or “Sensor calibration success.”
4. Override time sources by modifying RTC (real-time clock) values or intercepting GPS time sync functions.
5. Redirect logs to write into alternate locations, avoiding cloud synchronization or blackbox modules.
6. Where Syslog is used (in Linux-based IVIs), use tools like rsyslog with custom templates to generate false logs.
7. If log rotation is active, replace compressed logs with forged versions that pass hash validation.
8. Monitor OEM diagnostics tools to ensure these fake logs are displayed during analysis.
9. Test backup log restoration mechanisms to confirm your forged entries persist.
10. Use encryption to protect your forged logs from tampering by OEM recovery processes.

🛡️ Technique 3: IDS/Monitoring Bypass

Procedure 1: Mimic Normal CAN Traffic Patterns

1. Record hours of benign CAN traffic from the target vehicle using tools like CANalyzer or PCAN.
2. Analyze traffic to find standard frequency, arbitration ID timing, and signal patterns.
3. Develop malicious packets that mimic legitimate patterns—for instance, slowly changing speed values or blinking indicators.
4. Throttle the rate of injection to avoid detection by threshold-based IDSs.
5. Insert "heartbeat" messages that resemble OEM diagnostics tools to keep systems from flagging idle ECUs.
6. Randomize injection timings within bounds of normal behavior to evade time-based anomaly detection.
7. If the IDS monitors entropy, add noise-filled packets to blend in with normal bus chatter.
8. Test on actual hardware or digital twin to confirm alerts are not triggered.
9. Continue to update attack signatures based on OEM firmware changes.
10. Use bus-off simulation to temporarily disable IDS modules during the attack window.

Procedure 2: Bypass IVI System File Monitoring

1. Identify file integrity mechanisms used in IVI systems (e.g., Tripwire, AIDE, or custom OEM scripts).
2. Locate the hash database files and disable their update routines.
3. Modify monitored files (e.g., navigation software or API endpoints) with malicious scripts.
4. Recalculate and replace the hash entries with new forged hashes.
5. Patch the file integrity checker to always return clean results.
6. If hash validation is done remotely, intercept the validation calls and redirect them locally.
7. Simulate daily operation and validate that tampering is not flagged by OEM monitoring systems.
8. Introduce your backdoor or script-based payloads gradually to avoid suspicion.
9. Monitor system logs for attempted remote remediation or alerting.
10. Maintain persistence by embedding your checker bypass into the bootloader or init process.

#7 - 🔐 Tactic: Credential Access (ATA-007)

Objective: Gain unauthorized access to stored or transmitted credentials inside automotive systems to impersonate users or gain elevated control of ECUs, telematics units, or cloud services.

✅ Technique 1: Extract Credentials from Telematics Unit Firmware (ATA-007-T001)

📍 **Procedure 1:** Dump firmware and reverse-engineer credential storage

1. **Identify and acquire telematics ECU**: Gain physical access to the vehicle and locate the telematics control unit (TCU), typically under the dashboard or trunk. Remove it carefully to avoid triggering anti-tamper mechanisms.
2. **Connect to memory interface**: Use hardware tools like JTAGulator, Bus Pirate, or ChipWhisperer to identify debug or memory interfaces on the PCB (e.g., SPI flash, eMMC).
3. **Dump firmware**: Extract the full firmware image using tools like Flashrom or dd. Ensure integrity by checking hash consistency after multiple reads.

4. **Analyze filesystem**: Mount or unpack the firmware image. Look for embedded Linux systems (BusyBox, Yocto) and common partitions like /etc, /home, or /var.
5. **Search for credential files**: Look for config files (e.g., config.json, wpa_supplicant.conf, .env) that may store plain-text or base64-encoded secrets.
6. **Reverse-engineer credential logic**: If obfuscated, decompile the binaries using Ghidra or IDA Pro to find how credentials are handled or validated.
7. **Extract hardcoded keys**: Look for RSA private keys, JWT secrets, or OAuth tokens in strings or memory dumps.
8. **Attempt authentication**: Use extracted credentials to log into TCU web interfaces or backend services if reachable (e.g., OTA server, MQTT broker).
9. **Escalate access via privilege tokens**: If tokens are privileged, try accessing APIs like remote unlock, engine start, or vehicle tracking.
10. **Store credentials securely**: Avoid exposing or misusing them. If in a red team/PoC setup, document findings and report vulnerabilities responsibly.

📍 **Procedure 2:** Use UDS service to brute-force security access level

1. **Connect to CAN via OBD-II or Ethernet**: Tap into the vehicle network using a CAN interface (e.g., CANtact, ValueCAN) through the OBD-II port or vehicle gateway.
2. **Send UDS DiagnosticSessionControl requests**: Use tools like UDSim, Scapy, or Python-CAN to initiate diagnostic sessions (0x10 subfunctions).
3. **Issue SecurityAccess request (0x27)**: This service challenges the tester to respond with a seed-key authentication. Observe ECU's response to get the seed.
4. **Log seed values**: Capture several seeds from multiple requests. Some ECUs use predictable patterns or weak entropy.
5. **Analyze seed algorithm**: Reverse the firmware to find the seed-to-key algorithm or brute-force using rainbow tables.
6. **Send calculated key response**: Reply with the expected key (matching the received seed) within the session timeout window.
7. **Verify access granted**: A successful response from the ECU will confirm higher privileges (e.g., Level 2 access = developer mode).
8. **Read sensitive memory regions**: Use ReadMemoryByAddress (0x23) or ReadDataByIdentifier (0x22) to extract user tokens, Wi-Fi passwords, or calibration data.
9. **Check for brute-force limits**: Some ECUs implement lockout counters. Track responses and reset using power cycles if needed.

10. **Document results and test repeatability**: Validate if method works across vehicle models with same ECU supplier.

✅ **Technique 2: Intercept Vehicle-to-Cloud Token Exchange (ATA-007-T002)**

📍 **Procedure 1:** Set up MITM proxy to capture OAuth tokens

1. **Identify communication channels**: Vehicles typically use LTE, Wi-Fi, or Ethernet for telematics. Isolate the TCU and simulate a base station or AP to intercept traffic.
2. **Deploy LTE/Wi-Fi MITM**: Use tools like srsLTE (for LTE) or hostapd + mitmproxy (for Wi-Fi) to establish a rogue network the vehicle connects to.
3. **Spoof DNS or intercept traffic**: Redirect telematics domains (e.g., api.vehiclecloud.com) to attacker-controlled proxy using DNS spoofing or ARP poisoning.
4. **Capture HTTPS traffic**: Attempt to downgrade or intercept TLS. Use custom certificates or exploit TLS misconfiguration if certificate validation is weak.
5. **Inspect requests for OAuth headers**: Analyze captured API requests for Authorization headers (Bearer <token>).
6. **Extract access and refresh tokens**: Store tokens securely and replay them against cloud APIs or mobile apps.
7. **Check token scope**: Determine whether the token allows read-only access (e.g., vehicle status) or write privileges (e.g., door lock, geofencing).
8. **Access vehicle cloud console**: Use intercepted token to access real-time location, trip history, or vehicle control settings.
9. **Test revocation**: Log out the user from the official app and see if the token still functions, indicating a lack of server-side expiration.
10. **Document impact**: Show how token theft can lead to impersonation and remote vehicle control.

📍 **Procedure 2:** Exploit companion mobile app to retrieve stored credentials

1. **Download and decompile APK**: Use tools like APKTool or jadx to extract source code from the mobile app linked to the vehicle.
2. **Inspect authentication flows**: Identify how the app authenticates—OAuth, JWT, Basic Auth, or proprietary tokens.
3. **Look for hardcoded secrets**: Search the codebase for API keys, client secrets, or test credentials.

4. **Check SharedPreferences and local storage**: Use adb or root access to view stored login data, cookies, or cached tokens.
5. **Intercept traffic during login**: Use Burp Suite or mitmproxy to view login credentials in transit if TLS pinning is bypassed.
6. **Bypass TLS pinning**: Use Frida or Xposed modules to defeat certificate pinning, enabling full HTTPS visibility.
7. **Replay session**: Use captured tokens to replay sessions in Postman or curl, testing access scope.
8. **Check backend API documentation (if leaked)**: Some apps expose Swagger endpoints that can be abused with valid tokens.
9. **Trigger password reset**: Abuse insecure reset flows (e.g., no 2FA) to hijack accounts.
10. **Demonstrate full compromise**: Show how an attacker could impersonate a user, access car location, or trigger remote start.

✅ **Technique 3: Key Fob Signal Replay (RF Credential Theft)**

Procedure 1: Relay Attack Using Software-Defined Radios (SDRs)

1. **Select a target vehicle** that uses passive keyless entry (PKE) and start systems.
2. **Deploy two attackers**: one near the key fob (e.g., house door), and one near the car.
3. **Equip each attacker** with an SDR setup like HackRF or Yard Stick One for RF relay.
4. **Use open-source replay tools** (e.g., RollJam or custom GNU Radio scripts).
5. **The first attacker captures the fob signal** and relays it in real-time over Wi-Fi/Bluetooth.
6. **Second attacker transmits it near the vehicle**, tricking it into thinking the fob is nearby.
7. **Vehicle unlocks and allows engine start**, completing credential theft.
8. **Record RF waveforms** and timestamped logs for forensic replay or debugging.
9. **Test across vehicle brands**—some have rolling codes or nonce mechanisms that block replay.
10. **Analyze the effectiveness** and possible countermeasures (e.g., ultra-wideband tech, motion sensors).

🔁 **Procedure 2:** Capturing and Replaying Unlock Signals with Low-Cost RF Tools

1. **Select a target vehicle** that uses older or vulnerable RF key fob systems, especially models known to lack rolling code protections.
2. **Procure a low-cost RF transceiver**, such as the RTL-SDR, Flipper Zero, or CC1101 modules with Arduino. These can listen and transmit within the 300–433 MHz band.
3. **Observe the vehicle owner unlocking their car** in a parking lot, garage, or driveway—focus on moments where the fob is physically used.
4. **Record the RF signal** using your transceiver tool when the unlock button is pressed. Tools like Universal Radio Hacker or Audacity with RTL-SDR can help analyze the raw waveform.
5. **Analyze the signal** to confirm whether it uses fixed code or rolling code. If the code remains static upon repeated presses, it's vulnerable.
6. **Replay the captured signal** at a later time when the owner is away from the vehicle. Use the same device to broadcast the exact waveform.
7. **Observe vehicle unlocking**, confirming a successful replay attack. In many vulnerable systems, this alone grants full entry access.
8. **Test locking and engine start** capabilities if supported. Some cars separate ignition authorization from door unlock.
9. **Record all attempts and timestamps**, documenting which fobs and vehicle models are vulnerable. Compare success rates and distances.
10. **Optional: Extend the attack** by modifying the replayed signal slightly to test detection thresholds or interfere with keyless entry behavior.

#8 - 🔍 Tactic: Discovery (ATA-008)

Objective: Identify internal components, network topology, firmware versions, and connected modules within the vehicle's electronic architecture. This stage helps attackers map out the system before launching targeted attacks.

✅ Technique 1: ECU Enumeration via CAN Probing

📘 **Procedure 1:** Passive Bus Listening and ECU ID Mapping

1. Connect to the OBD-II port using a CAN interface tool like a CANtact or PCAN-USB.
2. Configure the tool to listen passively to all CAN traffic without transmitting any frames initially.
3. Start the vehicle to activate all ECUs and allow natural communication to begin.
4. Log all traffic, noting arbitration IDs (CAN IDs) and their frequency of transmission.

5. Identify which IDs are associated with different ECUs based on known ID mappings.
6. Use open-source tools like SavvyCAN or Wireshark (with CAN dissector) for better visualization.
7. Build a table of ECUs and corresponding CAN IDs from the captured data.
8. Compare against manufacturer documentation or reverse-engineered databases.
9. Correlate messages with physical actions (e.g., door open, throttle press) for better mapping.
10. Maintain stealth by avoiding any message injection that could trigger alerts or DTCs.

📘 **Procedure 2:** Brute Force ECU Discovery by Sending Tester Present

1. Connect a CAN injector device (e.g., ValueCAN, Arduino+CAN shield) to the OBD-II port.
2. Send "Tester Present" diagnostic request (0x3E) to a wide range of CAN IDs (e.g., 0x700–0x7FF).
3. Monitor for positive responses (e.g., 0x7E8, 0x7EA) from any responsive ECUs.
4. Log which IDs respond, indicating active ECU addresses.
5. Analyze response data for clues about ECU type, vendor, or function.
6. Adjust timing and frequency to avoid tripping diagnostic trouble codes (DTCs).
7. Use security access sequences to further fingerprint ECUs.
8. Compare responses to known UDS protocol behavior.
9. Compile an active ECU list based on successful responses.
10. Store response frames for offline analysis and mapping.

✅ **Technique 2: Network Segmentation Discovery**

📘 **Procedure 1:** Inject Messages to Identify Bridged Networks

1. Select a known message that's common to one network segment (e.g., infotainment).
2. Transmit that message onto a different segment (e.g., powertrain bus) using a gateway bypass.
3. Observe whether the message propagates or is blocked.
4. Repeat this process with varying priority IDs.
5. Log which messages successfully cross the segment and which don't.
6. Identify the gateway behavior (filtering, NAT, translation).

7. Map the internal network topology based on message flow.
8. Use this to identify isolated vs. bridged domains.
9. Look for relayed diagnostics requests across buses.
10. Use this map to identify vulnerable routes to safety-critical systems.

📘 **Procedure 2:** Request Routing Diagnostics via Gateway

1. Send UDS "Request Download" commands from one domain (e.g., telematics unit).
2. Observe if the request is routed by the central gateway to another ECU (e.g., braking system).
3. If successful, extract details of the destination ECU (e.g., firmware size, memory map).
4. Use this info to infer gateway routing rules.
5. Try routing requests from multiple origins to test access rules.
6. Log which domains can access which safety-critical ECUs.
7. Analyze if the gateway uses access controls or segmentation policies.
8. Attempt to bypass restrictions using message spoofing.
9. Use fingerprinting to locate weak segments.
10. Identify insecure domains that can reach high-trust zones (pivot potential).

✅ **Technique 3: Firmware Version Enumeration**

📘 **Procedure 1:** Query ECU Software Version via UDS

1. Connect a diagnostic tool to the vehicle via the OBD-II port.
2. Send UDS request 0x22 with the PID for software version (e.g., 0xF190).
3. Log the response containing the firmware version string.
4. Repeat for all active ECUs discovered earlier.
5. Compare firmware versions with known public advisories or recall databases.
6. Identify outdated ECUs with known vulnerabilities.
7. Cross-reference versioning with firmware update files (if leaked or available).
8. Build a version map of all modules in the vehicle.
9. Test if any ECUs respond to requests without authentication.
10. Use version info to plan downgrade or exploit-based attacks.

📘 **Procedure 2:** Extract Firmware Over-the-Air or via Update Media

1. Obtain a dealer update USB stick, SD card, or OTA update file for the vehicle model.
2. Analyze the update files on a workstation using hex editors or binwalk.
3. Identify embedded firmware images and their associated version metadata.
4. Cross-reference image signatures with in-vehicle firmware version responses.
5. Decrypt or decompress firmware using vendor-specific tools or leaked scripts.
6. Locate memory regions, bootloaders, or update modules in the firmware dump.
7. Extract diagnostic strings or debug symbols for additional ECU info.
8. Match extracted versions with on-vehicle responses to validate accuracy.
9. Use this firmware intelligence to build exploit payloads or spoof updates.
10. Develop tools for future automation of version enumeration and vulnerability matching.

#9 - Tactic: Lateral Movement (ATA-009)

Objective:
Gain access to additional vehicle ECUs or subsystems after an initial compromise by pivoting through in-vehicle networks (CAN, LIN, Ethernet) or wireless interfaces (Wi-Fi, Bluetooth, V2X), enabling further control or data extraction beyond the initially compromised module.

Technique 1: Pivot Through Infotainment System to Telematics Unit

Procedure 1: Exploit media playback buffer overflow

1. Identify the vehicle's media processor or infotainment SoC model and firmware.
2. Develop a malicious audio/video file exploiting buffer overflow in the parser.
3. Load the file onto a USB stick or stream via Bluetooth.
4. Monitor infotainment system behavior after playback starts—look for crashes or reboots.
5. Confirm code execution via injected network packets or file write operations.
6. Use the compromised infotainment system to scan the internal Ethernet or CAN network.
7. Identify the IP address or CAN arbitration ID of the telematics control unit (TCU).
8. Craft packets emulating legitimate diagnostic traffic to the TCU.

9. Exploit any known vulnerability (e.g., open debug port) in TCU firmware.
10. Establish persistent access to TCU, enabling long-term tracking or command injection.

Procedure 2: Use infotainment debug shell for pivot

1. Acquire firmware image of infotainment system from online update repositories.
2. Analyze filesystem to locate hidden debug shells or SSH daemons.
3. Recreate the environment using a virtual machine or SoC emulator.
4. Re-enable debug access via crafted USB or Wi-Fi command injection.
5. Gain shell access and enumerate running services, looking for hardcoded credentials.
6. Identify open network interfaces and routes to TCU or other ECUs.
7. Use standard tools (e.g., netcat, nmap) to scan reachable IPs.
8. Interact with telematics interface via REST API or raw socket connection.
9. Escalate privileges by triggering a vulnerable service or exploiting an insecure default configuration.
10. Establish a secure reverse tunnel to attacker infrastructure via LTE or Wi-Fi.

Technique 2: Relay Diagnostics via Gateway Bypass

Procedure 1: Replay authorized UDS commands

1. Capture legitimate diagnostics session between OEM tool and vehicle.
2. Save UDS messages for commands like readDTC, routineControl, and writeMemory.
3. Use a CAN interface (e.g., CANtact, ValueCAN) to replay these messages from another ECU or port.
4. If gateway filtering is active, inject traffic from an ECU on the same CAN domain.
5. Replay writeMemory to modify ECU calibration settings or memory regions.
6. Trigger software upload mode to force firmware flash.
7. Confirm whether gateway logs unauthorized activity or allows pass-through.
8. Adjust CAN arbitration ID and timing to avoid detection.
9. Record successful payload execution or ECU reboot behavior.
10. Use this pathway to move laterally into safety-critical domains.

Procedure 2: Modify OBD-to-Gateway whitelist

1. Identify the gateway ECU and extract firmware (JTAG or UDS memory read).
2. Locate access control list or function selector governing OBD port behavior.
3. Modify the firmware locally to whitelist non-OEM tools or remote IPs.
4. Reflash gateway ECU using bench flashing tool or via OBD if write access is allowed.
5. Connect custom diagnostic tool via Bluetooth OBD dongle.
6. Send gateway commands that redirect traffic to internal powertrain network.
7. Access ECUs not normally reachable from external ports.
8. Execute service routines on ECUs like power steering, braking, or airbags.
9. Dump memory or configuration files from the new targets.
10. Log all lateral hops and persist access for future OTA-triggered commands.

Technique 3: Exploit V2X or Bluetooth for Cross-ECU Control

Procedure 1: Abuse Bluetooth Hands-Free Profile

1. Scan for Bluetooth services exposed by vehicle's infotainment or telematics module.
2. Connect to the Hands-Free Profile (HFP) and initiate malformed AT commands.
3. Trigger buffer overflow or stack corruption on vulnerable firmware.
4. Inject shell payload or establish a command channel through RFCOMM.
5. Explore local filesystem for inter-ECU bridge configuration files.
6. Send crafted D-Bus messages or gRPC calls to telematics or sensor ECUs.
7. Dump keys or certificates used for secure updates or vehicle control.
8. Emulate authenticated user behavior via modified Bluetooth packet injection.
9. Relay commands to more privileged ECUs (e.g., BCM, VCM).
10. Use Bluetooth as a pivot point for remote control or surveillance.

Procedure 2: V2X replay attack from nearby RSU emulator

1. Set up a rogue Road Side Unit (RSU) emulator with spoofed OEM credentials.
2. Broadcast V2X packets containing alerts or commands (e.g., emergency braking).
3. Capture and replay packets that trigger inter-ECU messages (e.g., to braking module).
4. Modify timestamps and message IDs to bypass replay protection.
5. Confirm packet acceptance and ECU reaction via sniffer on CAN/Ethernet.

6. Repeat attack across multiple ECUs to evaluate cross-domain movement.
7. Check if the vehicle firmware isolates V2X messages by gateway or firewall.
8. If not, inject payloads via repeated DSRC or C-V2X frames.
9. Trigger unintended behaviors like flashing hazards, adjusting mirrors, or silencing alerts.
10. Use the open V2X channel for continued access even after ignition off.

#10 - 🚩 Tactic: Collection (ATA-010)

Objective: Gather sensitive in-vehicle data (e.g., location history, sensor logs, personal data) that may be used for further exploitation, surveillance, or resale.

🔧 Technique 1: ATA011-T1 – Collect Navigation History

✅ **Procedure 1:** Extract GNSS and trip data from infotainment system

1. Connect to the vehicle's OBD-II port or Ethernet debug port using a compatible adapter.
2. Access the vehicle’s internal infotainment system running QNX, Android Auto, or similar.
3. Use diagnostic tools (e.g., D-Bus or QNX Momentics) to explore mounted storage paths.
4. Locate GNSS logs or SQLite databases that track past routes, timestamps, and waypoints.
5. Use forensic tools to extract and decode the GPS track points from binary or DB format.
6. Visualize data using mapping software like GPSVisualizer or Google Earth for route replay.
7. Note if the data contains Wi-Fi locations, search history, or POI entries.
8. Correlate GPS data with driver profiles, time, or recurring locations (e.g., home, office).
9. Attempt to extract backup data via USB interface (used for OTA updates or service logs).
10. Confirm if the system purges location history after logout or if it remains persistent.

✅ **Procedure 2:** Pull navigation cache from mobile-connected systems

1. Pair a smartphone to the car using Bluetooth or USB and launch Android Auto or CarPlay.
2. Simulate regular usage (navigation to multiple locations) and disconnection.
3. Access the mobile device's cache and logs using **adb** or forensic apps like Cellebrite.
4. Locate cached maps, voice commands, or GNSS logs stored by navigation apps.
5. Extract trip history, search queries, and estimated arrival/departure data.
6. Attempt reverse-linkage to car identity (VIN or driver profile) via synced metadata.
7. If cloud sync is enabled, intercept outbound traffic to services like Google Timeline.
8. Use DNS sniffing or SSL stripping tools in a controlled network to observe sync patterns.
9. Combine mobile and vehicle-side data to reconstruct full travel profiles.
10. Analyze whether a full factory reset removes this data or if traces persist.

🔧 **Technique 2: ATA011-T2 – Collect Sensor/Camera Logs**

✅ **Procedure 1:** Tap CAN bus for logged events and sensor data

1. Connect to the vehicle's CAN bus via the OBD-II port or direct wiring under the dash.
2. Use a CAN interface (e.g., CANtact, ValueCAN, or USB2CAN) with open-source software like SavvyCAN.
3. Begin passive logging during vehicle motion to record camera triggers and sensor alerts.
4. Identify camera-based object detection packets by observing message frequency and ID.
5. Isolate recurring CAN IDs linked to lane detection, obstacles, and blind spot activity.
6. Correlate these packets with timestamps and known driving conditions (day/night, traffic).
7. Create a timeline of driver interactions and sensor events (braking, swerving, alerts).
8. Save the data logs in a binary or DBC-decoded format for analysis and replay.
9. Attempt re-injection of packets to test system response in simulated environments.
10. Document whether logged data contains metadata revealing image type or event classification.

✅ **Procedure 2:** Access camera logs via Ethernet debug port

1. Open the vehicle's service panel to access the Ethernet debug port (100BASE-T1).
2. Connect using an automotive Ethernet adapter (e.g., Intrepid RAD-Moon).
3. Initiate communication with ECU hosting camera input—typically ADAS or IVI modules.
4. Use packet sniffing tools like Wireshark with correct VLAN ID and filters for camera traffic.
5. Capture data streams and logs used for object detection, collision warnings, etc.
6. Decode traffic containing camera metadata, timestamps, and classification labels.
7. Attempt extraction of actual image frames or motion vector logs if not encrypted.
8. Analyze log timestamps and correlate them with physical driving behaviors.
9. Identify frequency of detection failure or misclassification logs.
10. Determine whether logs are stored persistently or flushed after shutdown.

🔧 **Technique 3: ATA011-T3 – Collect User Behavior & Biometrics**

✅ **Procedure 1:** Monitor driver profiles stored in head unit

1. Access the IVI (infotainment) system via touchscreen, debug menu, or backend shell.
2. Explore user profiles that store seat position, mirror adjustments, voice print data.
3. Look for biometric access data (e.g., facial recognition, fingerprint scan traces).
4. Extract profile storage directories and export them via USB or network.
5. Decode proprietary formats or use OCR/ML models to analyze stored face image templates.
6. Investigate logs showing frequency of use per profile and switching behavior.
7. Correlate biometric data with times of use, environmental lighting, or failure cases.
8. Attempt profile spoofing or injection to test unauthorized behavior control.
9. Check for linked phone numbers, call history, or contact book stored in profile.
10. Assess whether deleting a profile removes all biometric and usage data securely.

✅ **Procedure 2:** Exploit driver monitoring system (DMS) logs

1. Identify the in-cabin camera (typically IR-based) used for driver monitoring (drowsiness, attention).
2. Connect to the ECU managing DMS, often shared with ADAS or IVI module.

3. Access stored logs containing driver eye movement, head tilt, and gaze vector.
4. Download raw log files, often stored in proprietary or CSV format.
5. Use visualization tools to map driver alertness against time and route segments.
6. Attempt to extract frames of eye or face snapshots for biometric profiling.
7. Investigate how frequently the system logs "eyes off road" or "drowsy" states.
8. Cross-reference with road type, driving speed, or time of day for pattern analysis.
9. See if the DMS uses cloud analytics—intercept outbound logs if enabled.
10. Assess whether disabling DMS (e.g., via UI or OBD command) halts log generation.

#11 - 🎯 Tactic: Command and Control (ATA-011)

Objective: To establish a communication channel between the compromised vehicle and an external attacker, enabling remote command execution, data exfiltration, or further control actions.

🛠️ Technique 1: ATA-011A – Cellular-based C2 Channel

📋 **Procedure 1:** Hijack Telematics SIM Module to Receive Commands

1. Identify the telematics control unit (TCU) in the vehicle that houses the embedded SIM used for LTE/3G connectivity.
2. Remove or intercept the SIM physically or virtually through firmware extraction or remote compromise.
3. Analyze APN settings, DNS queries, and IMSI info to determine the network provider and existing C2 infrastructure.
4. Replace SIM or spoof it using a programmable SIM card (e.g., SIMtrace or Osmocom SIM).
5. Set up a rogue base station or redirect DNS to a malicious C2 server under attacker control.
6. Configure the vehicle's TCU to accept and initiate outbound requests to your C2 domain/IP.
7. Use HTTP(S), MQTT, or proprietary protocols to send commands disguised as legitimate telemetry data.
8. Issue commands to modify vehicle state: disable GPS, spoof speed, or adjust diagnostics.

9. Ensure persistence by modifying the TCU's firmware or bootloader to retain your network config after reboot.
10. Monitor traffic continuously and log responses to validate successful command execution.

📋 **Procedure 2:** Exploit OTA (Over-the-Air) Channel for Remote Control

1. Identify the OEM's OTA update platform used to deliver firmware and configuration changes to vehicles.
2. Perform DNS spoofing or SSL stripping to intercept OTA traffic if not fully encrypted or validated.
3. Capture and analyze update payloads, headers, and authentication tokens.
4. Create a malicious update package that includes embedded C2 stubs or scriptable backdoors.
5. Host the fake OTA update on a domain resembling the OEM's legitimate update server.
6. Redirect vehicle update requests to the malicious server using rogue Wi-Fi, cellular manipulation, or DNS hijack.
7. Once the update is installed, gain a callback to your C2 server with privilege inside the ECU.
8. Use the backdoor to send periodic beacons and accept remote commands over HTTPS.
9. Validate control by issuing minor diagnostic commands first, like resetting fault codes or toggling sensors.
10. Maintain stealth by limiting update frequency or mimicking normal update sizes and timestamps.

🛠️ **Technique 2: ATA-011B – Wi-Fi or Bluetooth C2 Channels**

📋 **Procedure 1:** Establish Covert Bluetooth Channel via Infotainment

1. Pair a malicious smartphone with the vehicle's infotainment system through Bluetooth.
2. Reverse engineer the infotainment Bluetooth services using tools like obex, rfcomm, or btmon.
3. Identify insecure services (e.g., phonebook, audio streaming, tethering) that allow unsolicited data transfer.

4. Abuse tethering to give the infotainment unit internet access or leak data via DUN profiles.
5. Install a hidden app on the paired phone that acts as a relay to an attacker-controlled server.
6. Create a loop where commands from the attacker are sent to the app and relayed to the infotainment unit via Bluetooth.
7. Trigger actions in the infotainment system—e.g., file writing, media injection, or executing malformed packets.
8. Observe if the infotainment OS exposes shell access, local service calls, or debug ports.
9. Maintain Bluetooth pairing persistence by modifying trusted device settings on the head unit.
10. Hide indicators from the driver by suppressing notifications or toggling visibility off using developer menus.

📋 **Procedure 2:** Wi-Fi C2 via Malicious Hotspot

1. Identify vehicles that auto-connect to known Wi-Fi networks, including open SSIDs or OEM hotspots.
2. Set up a rogue access point with the same SSID as a known trusted network (e.g., "ToyotaWiFi").
3. Wait for the vehicle to auto-connect or entice users by offering firmware updates via a captive portal.
4. Monitor DHCP logs to get the vehicle's MAC/IP and identify traffic to vulnerable internal services.
5. Host a small C2 server within the LAN subnet that can issue commands via known infotainment APIs.
6. Send WebSocket or REST API calls to infotainment ports exposed within the Wi-Fi subnet.
7. Exploit services like D-Bus, ADB over TCP, or debug HTTP servers on port 8080.
8. Establish persistence by injecting startup scripts or modifying local config files over Wi-Fi.
9. Periodically re-broadcast your rogue hotspot to catch reconnections.
10. Exfiltrate data like GPS logs or media library metadata back through the Wi-Fi channel.

🛠️ Technique 3: ATA-011C – Vehicle-to-Vehicle (V2V) or V2X C2 Channel

📋 **Procedure 1:** Use DSRC/V2X Messages for Covert Commands

1. Identify whether the target vehicle supports V2V or V2X (Vehicle-to-Everything) communication using DSRC or C-V2X.
2. Monitor for periodic beacons or Basic Safety Messages (BSMs) broadcast from nearby cars.
3. Create or modify a V2X device (e.g., using Cohda Wireless or open C-V2X stack) to craft custom messages.
4. Embed command instructions within optional V2X payload fields, abusing extended headers or misused fields.
5. Transmit these messages within the expected broadcast interval to avoid detection.
6. Watch the target vehicle's CAN traffic or debug logs to confirm receipt of V2X messages.
7. Trigger specific events based on V2X input—like lane change alerts, proximity beacons, or emergency vehicle override.
8. Inject crafted messages to simulate road hazards, spoof traffic signals, or trigger evasive behavior.
9. Record any unexpected system behavior or service crashes caused by malformed V2X data.
10. Loop this mechanism to keep sending V2X commands from multiple attacker vehicles (relay-style C2).

📋 **Procedure 2:** Covert C2 via Roadside Infrastructure (RSU Exploit)

1. Locate and compromise a Roadside Unit (RSU) that's part of a Smart City V2X system (e.g., traffic light controller).
2. Analyze firmware or exposed ports of RSU using Wireshark, Telnet, or local debug UART access.
3. Modify RSU firmware or configuration to start broadcasting attacker-crafted V2X messages.
4. Ensure messages are formatted correctly and meet V2X authentication criteria (or bypass them).
5. Time the transmission to overlap with known vehicle passing windows (e.g., during red light).
6. Send payloads that mimic legitimate advisories (e.g., "accident ahead," "reroute") but include embedded commands.

7. Use vehicle logs or traffic flow change to confirm command reception.
8. Alter RSU behavior to issue commands periodically, maintaining a stable C2 window.
9. Persist your control by locking RSU firmware or reusing default credentials post-reboot.
10. Monitor multiple vehicles simultaneously for broader attack surface and data collection.

#12 - 🎯 Tactic: Exfiltration (ATA-012)

Goal: To remove data from the in-vehicle network or telematics systems and transfer it to an external actor without detection. This can include exfiltrating GPS logs, user preferences, driving behavior, sensor data, or firmware files over cellular, Wi-Fi, or physical means.

✅ Technique 1: Exfiltration Over Cellular Network

ID: ATA-012-T001

▶️ **Procedure 1:** Abuse Telematics SIM for Data Exfil

1. Gain access to the telematics control unit (TCU) using previous attack vectors (e.g., diagnostic injection).
2. Establish a shell or script inside the TCU OS (commonly Linux-based).
3. Identify SIM configuration using mmcli or equivalent modem management tools.
4. Modify or piggyback on existing APN settings to allow external IP-based exfil.
5. Compress or encrypt sensitive data (e.g., location logs) using built-in tools like tar or openssl.
6. Initiate periodic uploads using curl or wget to an attacker-controlled HTTPS endpoint.
7. Use random jitter in timing to evade behavioral anomaly detection.
8. Confirm success by validating receipt of files on remote server.
9. Monitor data usage to avoid tripping SIM data caps or fleet monitoring alerts.
10. Set cleanup logic in cron jobs to auto-delete logs after transfer.

▶️ **Procedure 2:** Create Reverse Tunnel via LTE Modem

1. Exploit OTA or exposed API to run custom daemon in vehicle's TCU.
2. Install lightweight tunneling tool like ngrok, autossh, or FRP (Fast Reverse Proxy).

3. Use credentials to open a reverse SSH tunnel from the car's LTE connection to an attacker server.
4. Tunnel diagnostic ports (e.g., TCP 23, 5555) for shell access.
5. Route exfiltrated data (e.g., CAN logs, user profiles) through the encrypted channel.
6. Validate low latency and throughput to support real-time stream if needed.
7. Implement auto-reconnect and keep-alive to persist across vehicle reboots.
8. Use DNS tunneling fallback if HTTPS tunnel is blocked.
9. Log all outbound connections and filter noise from TCU system logs.
10. Shut down tunnel after exfiltration to remain stealthy.

✅ **Technique 2: Exfiltration Over Physical Medium**

ID: ATA-012-T002

▶️ **Procedure 1:** Store Data on USB Plugged into Infotainment

1. Plug in a malicious USB drive into infotainment or rear-seat media ports.
2. Exploit autorun (if enabled) or use pre-scripted file system triggers to launch code.
3. Mount internal partitions readable by media player (e.g., /media/internal).
4. Copy desired files like drive_log.csv, userdata.db, or nav_cache.sqlite.
5. Store files in hidden USB directories or disguise them as common media files.
6. If allowed, re-encrypt using AES with pre-shared key stored on USB.
7. Eject USB or wait until the attacker retrieves the vehicle.
8. Confirm whether files are still cached on internal drive after USB removal.
9. Inspect infotainment logs to check for evidence of file access.
10. Test persistence—whether repeat insertions trigger auto-exfil.

▶️ **Procedure 2:** Hidden SD Card Logger Behind Dashboard Panel

1. Gain interior access (valet mode, service bay, or physical break-in).
2. Locate a diagnostic port or spare connector near the head unit or ECU.
3. Attach a Raspberry Pi or ESP32 with a microSD logger and minimal power draw.
4. Connect to CAN or LIN bus using appropriate transceivers.
5. Record traffic and dump into structured files per timestamp.
6. Exfiltrate GPS coordinates, speed, braking patterns, or raw frame logs.

7. Optionally compress data on-device to extend storage life.
8. Ensure system does not draw excess power when vehicle is off.
9. Retrieve hardware after time delay (e.g., during next inspection).
10. Analyze recovered SD card contents for campaign success.

✅ **Technique 3: Exfiltration Over Wi-Fi**

ID: ATA-012-T003

▶️ **Procedure 1:** Use Drive-By Wi-Fi Upload

1. Exploit an open or weakly secured hotspot created by the vehicle (for updates or user connectivity).
2. Inject a script into infotainment or modem firmware to store sensitive logs.
3. As car approaches known Wi-Fi hotspots (e.g., attacker-controlled roadside APs), trigger uploads.
4. Use scp, rsync, or custom UDP beacon to send packets silently.
5. Include authentication tokens or signed headers to verify target server.
6. Throttle upload rate to avoid stalling other system tasks.
7. Log successful transmissions for later retrieval.
8. Randomize filenames or obfuscate metadata to avoid IDS flags.
9. Track GPS to identify repeated upload zones.
10. Clean up scripts after transfer or reboot to avoid forensic evidence.

▶️ **Procedure 2:** Hijack Wi-Fi Module for Passive Beacon Dump

1. Use shell access to scan nearby APs and connect to the attacker's.
2. Enable promiscuous or monitor mode on the vehicle's Wi-Fi chip (e.g., Broadcom/Qualcomm).
3. Capture beacons, probe requests, or passive channel data.
4. Dump captured logs periodically into local file system.
5. Once a pre-defined SSID appears (e.g., attacker base station), initiate data burst.
6. Transmit files using HTTP POST or custom UDP stream to avoid signature detection.
7. Optionally spoof MAC address per session to avoid profiling.
8. Disable Wi-Fi module after each burst to reduce visibility.

9. Confirm data integrity using hash comparison.
10. Leave dummy traffic or decoys in logs to mislead forensic analysis.

#13 - 🎯 Tactic: Affect Vehicle Function (ATA-013)

Goal: To intentionally degrade, alter, or disable the core mechanical or electronic functionality of the vehicle. This includes manipulating braking, steering, powertrain, charging, HVAC, or dashboard components—either subtly or catastrophically—to cause unsafe behavior, immobilize the car, or deceive the driver.

✅ Technique 1: Brake Manipulation via CAN Injection

ID: ATA-013-T001

▶️ **Procedure 1:** Disrupt Brake Signal from ABS ECU

1. Connect to the vehicle's CAN bus through OBD-II or direct wiring under the dashboard.
2. Identify the CAN ID used by the ABS ECU to transmit brake pedal or pressure signals.
3. Use a tool like a CAN injector (e.g., CANtact, ValueCAN) to replay known-good brake messages at a faster frequency.
4. Override the legitimate ABS module messages by creating a flood of spoofed ones, using the same CAN ID.
5. Monitor the vehicle to check if the ABS controller's inputs are ignored or replaced.
6. In low-speed tests, observe the brake lights and mechanical braking behavior (e.g., delays or failure to engage).
7. If functional, escalate to higher-speed controlled track tests to analyze vehicle response under dynamic loads.
8. Log all injection events and system faults from diagnostic tools.
9. Check dashboard alerts—some ECUs may detect checksum errors or out-of-range values.
10. Verify whether the driver receives any meaningful warning or if braking silently degrades.

▶️ **Procedure 2:** Force Brake Actuation During Driving

1. Use prior access (e.g., via gateway compromise) to transmit actuator commands to the brake ECU.
2. Inject a command at runtime that mimics a brake apply signal, especially in ADAS-enabled vehicles.
3. Craft your payload with matching arbitration ID and expected byte structure (reverse-engineered).
4. Replay this command during low-speed movement or at stops to validate effect.
5. Escalate testing to inject the command while the car is at cruising speed (~30–60 km/h).
6. Monitor for ABS/ESC intervention or any self-protection logic.
7. Observe driver reaction, system logs, and telemetry (e.g., G-forces from sudden deceleration).
8. If vehicle supports remote telemetry, confirm that OEM logging doesn't capture the injected command.
9. Attempt persistence by creating periodic triggers on ignition or GPS-based geofence.
10. Demonstrate repeatability of brake actuation without user input.

✅ **Technique 2: Disable Electric Power Steering (EPS)**

ID: ATA-013-T002

▶️ **Procedure 1:** Send Fault Message to EPS Controller

1. Connect to vehicle CAN bus with elevated access (e.g., through infotainment vulnerability or compromised gateway).
2. Identify EPS controller's arbitration ID and expected fault message format.
3. Craft a CAN frame that mimics an internal diagnostic fault—e.g., motor overcurrent or temp shutdown.
4. Inject the frame into the bus during normal driving.
5. Monitor for immediate power steering loss or increased steering resistance.
6. Capture dashboard warnings such as "Power Steering Fault" or "Service Steering System".
7. Observe driver reaction and test if manual steering effort becomes excessive at low speeds.
8. Verify if fault resets on reboot or requires dealer tools for clearing.
9. Replay fault message after each power cycle to simulate a persistent defect.

10. Log internal DTC codes pulled from the EPS module post-attack.

▶️ **Procedure 2:** Starve EPS of Voltage via Relay Control

1. Locate and access the relay or fuse box responsible for EPS power delivery (usually under hood or dashboard).
2. Install a custom relay interrupter—microcontroller-based or physical switch—that can toggle 12V supply.
3. Command the device remotely via BLE, RF, or onboard timers.
4. While the vehicle is moving slowly (e.g., parking), cut the power line to EPS.
5. Observe complete or partial loss of steering assistance.
6. If equipped, confirm loss via steering torque sensors or CAN feedback from EPS ECU.
7. Restore power to simulate an intermittent fault—ideal for confusing diagnostics.
8. Correlate power cuts with CAN message silence from EPS module.
9. Inspect system logs for undervoltage, CAN timeout, or watchdog reset events.
10. Evaluate physical safety implications when driver attempts quick maneuvers.

✅ **Technique 3: Manipulate Instrument Cluster/Dashboard Display**

ID: ATA-013-T003

▶️ **Procedure 1:** Spoof Speedometer and Warning Lights

1. Use a CAN interface to locate the arbitration ID responsible for speed, RPM, fuel, and warning indicators.
2. Disconnect or suppress messages from actual vehicle sensors (e.g., wheel speed sensors).
3. Inject your own messages to simulate vehicle status—e.g., 120 km/h when stationary.
4. Flash warning lights such as oil, engine, or brake even if conditions are nominal.
5. Test both analog-style and digital clusters for response accuracy and display latency.
6. Observe if spoofed data is reflected in HUDs or center infotainment panels.
7. Record driver confusion and behavioral response (e.g., pulling over due to fake alerts).
8. Confirm whether the system stores fake values in logs or resets on reboot.
9. Use this method as part of a distraction attack in coordination with ADAS faults.

10. Replay multiple speed profiles while vehicle is parked to simulate phantom behavior.

▶ **Procedure 2:** Force Cluster Reboot or Blackout

1. Identify the CAN IDs or Ethernet API responsible for cluster software updates or reboots.
2. Send a crafted payload that mimics a diagnostic session trigger (e.g., UDS service 0x11 for ECU reset).
3. Force a reboot during vehicle ignition or movement to disrupt visual feedback.
4. Observe cluster blackout, flickering, or total failure to boot into the GUI.
5. In electric vehicles, check if this impacts range display, battery level, or gear indicator.
6. Monitor CAN logs to verify absence of cluster messages (heartbeat, warnings).
7. Evaluate how long the reboot lasts and whether it recovers cleanly.
8. Chain this with ADAS attacks to amplify confusion during real-time driving.
9. Try different intervals (e.g., reboot every 60 seconds) to test resilience.
10. Confirm whether OEM anti-brick or failsafe systems kick in after repeated failures.

- Made by Sarthaka
  Subhankara Singh
  Intern id – 438
  Digisuraksha Parhari
  Foundation