Fast and Compromised: Automotive Security on a Timer

Enabling vehicle theft and remote control of vehicle functions in a limited time

ESCAR USA, May 2025

Omer Ziv, Erez Alfasi - Security Researchers, PlaxidityX

KEYWORDS: Penetration Testing, Reverse Engineering, Vehicle Level Testing, Car Hacking, CAN Injection, MQTT, Security Access, Secure Boot.

1 Abstract

This work presents a real-world penetration test of a production-ready electric vehicle, demonstrating full compromise within eight working days. Starting from minimal access via an insecure WiFi network, we achieved remote control over safety-critical systems and vehicle theft. Key issues included hardcoded credentials, insecure MQTT communication, and a weak UDS Security Access implementation. Our findings highlight the feasibility of rapid, high-impact attacks and underscore the need for secure boot, proper credential management, and robust in-vehicle network protections.

2 Introduction

In the automotive industry, it is commonly assumed that potential attackers require a long time to severely impact a vehicle. Vulnerabilities typically exploited in complex attacks include memory corruptions (buffer overflow, integer overflow, etc.), exposed APIs, side-channel attacks, and more. Executing these exploits as part of a complete attack scenario surely requires advanced expertise and time for research and implementation.

But is this the case? Are our cars that protected? Is it possible to create a serious safety-critical impact with less effort? This is a story of a race against time to penetrate a vehicle and achieve the most severe possible outcome. Two primary objectives were defined, representing some of the worst-case scenarios of vehicle hacking. The first was car theft (i.e., stealing the vehicle without a user key), and the second was remote compromise (i.e., affecting safety-critical functions such as steering and braking through remote access).

The talk presents a real-world vehicle penetration testing (PT) project conducted by our research team on an electric vehicle (EV) just before the start of production (SoP) as part of the regulatory process (ISO-21434: Road vehicles - Cybersecurity). During the span of eight working days, two researchers penetrated the vehicle system via WiFi and uncovered several critical security issues. Starting with minimal access via the WiFi interface—protected only by a weak default password—we were able to authenticate and perform lateral movement within the internal network, eventually reaching the vehicle's central gateway. From there, we replaced proprietary binaries with malicious versions to gain access to the CAN bus and control vehicle functions. Additionally, by reverse engineering internal binaries, we extracted hardcoded credentials for a secured MQTT server used for communication between the vehicle and the end-user mobile application, allowing us to perform all actions normally available to the user through the official app. During this process, we also recovered the Security Access algorithm used for UDS authentication, effectively bypassing the challenge-response mechanism and enabling unrestricted diagnostic access.

Through these combined actions, we achieved remote access to safety-critical functions while the vehicle was in motion, leading to a complete disruption of its operation with serious safety implications. Recovery required a full battery reset.

This talk outlines the procedures, discoveries, and implications of our research, highlighting how an attacker can achieve full compromise, even within a short timeframe. Some details have been redacted to preserve the confidentiality of the involved parties.

3 Laying the Groundwork: Understanding the Vehicle and the Potential Entry Points

The target was an EV tested shortly before the SoP. The vehicle's architecture included two main ECUs. The first is a Main Gateway, which connects to various CAN (Controller Area Network) bus interfaces managed by a classic AUTOSAR core and to Ethernet interfaces managed by a Linux core. The second is a High-Performance Computer (HPC) that consists of three components: Telematics Control Unit (TCU, equipped with an eSIM modem), In-Vehicle Infotainment (IVI, powered by an Android core), and Instrument Cluster (IC). Additionally, the gateway includes an On-Board Diagnostics (OBD) connection, which features Ethernet pins.

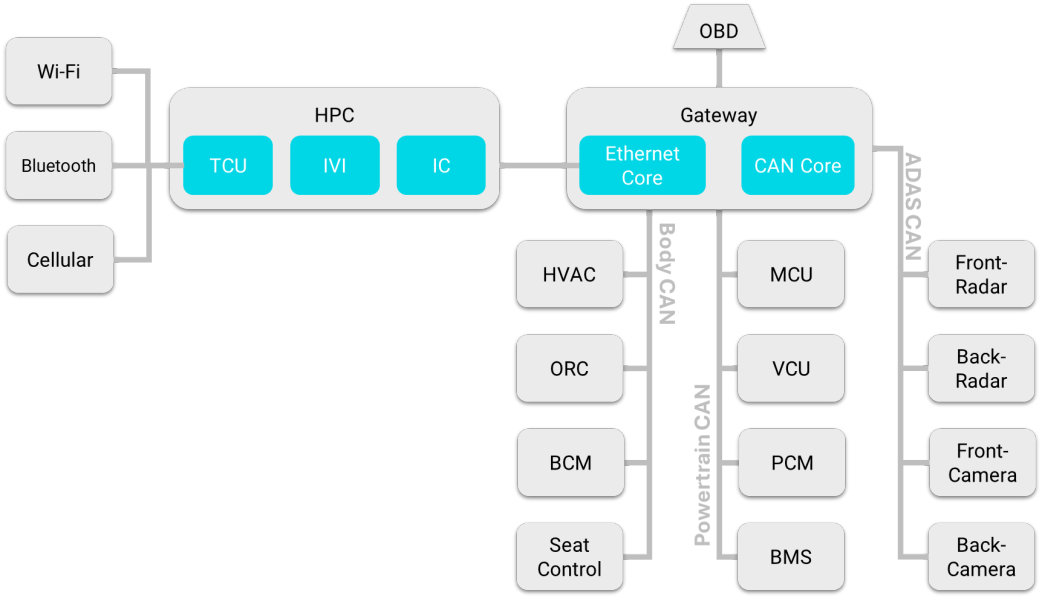

Figure 1. Partial vehicle-level architecture

Approaching the vehicle, we discovered a WiFi hotspot. Brute-forcing the WiFi password revealed that a default password is in place - ‘password123’. We connected to the WiFi network and scanned for IP addresses and open ports. The scan showed a critical and fundamental issue: an unprotected ADB (Android Debug Bridge) connection, which allowed direct access to a debug interface on the IVI with root privileges. Note that even if the attacker encounters a vehicle without a brute-forceable password, we discovered - surprisingly - that the same open ADB port is accessible through the OBD Ethernet connection, which is always available in the vehicle. Unfortunately, open ADB connections

in production vehicles are more common than one might think, as we encountered in several previous automotive PT projects.

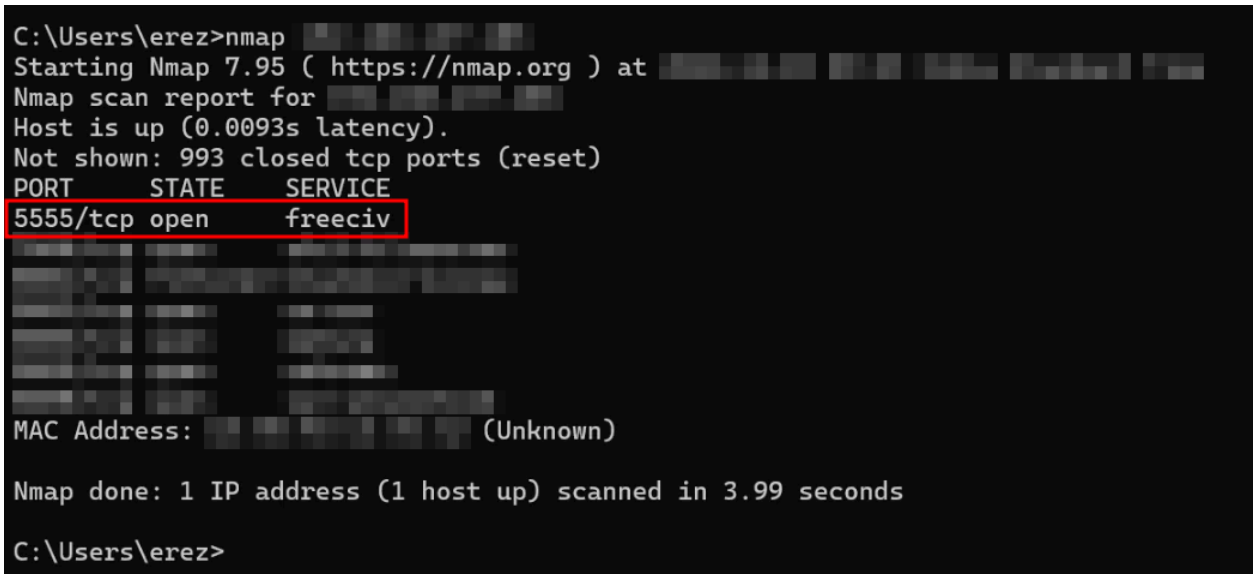

Figure 2. Unprotected ADB connection granting root access to the IVI through the WIFI. The same ADB port was also accessible through the always-available OBD Ethernet connection.

Further analysis revealed that the IVI had unrestricted internet access and lacked proper filtering mechanisms. With this access, we were able to obtain a remote shell to the IVI. Additionally, we found a persistent folder with read-write-execute permissions. This allowed us to upload a script that executed during boot, establishing a reverse shell from the vehicle back to our cloud machine. The absence of secure boot checks will enable us later to gain persistent code execution in the relevant area. This access allowed us to advance toward both objectives: stealing and compromising the vehicle. From that point forward, we no longer required proximity to the vehicle, enabling us to act as a remote persistent threat to the vehicle.

4 Breaking Through: Controlling the Gateway

At this point, we have already established a remote shell from our server to the IVI. To advance our attack, we aimed to gain access to the gateway, which is connected to all main interfaces. A scan from the IVI revealed an open SSH port on the gateway, which we attempted to access. We were then prompted to enter a password. Nmap scan revealed a probable kernel version, which allowed us to infer the likely Linux distribution. Using the last, we found the default root password through a quick online search. This enabled us to tunnel the communication and establish a direct connection between our remote server and the gateway's Linux core, granting a remote shell.

Exploring the gateway's Linux core file system, we found a persistent storage location with read-write-execute permissions (another example of missing secure boot and integrity controls). This directory contained proprietary scripts and binaries triggered by systemd jobs during boot. By replacing one of these binaries with our malicious version, we were able to execute custom binaries, achieving code execution on the gateway. Once we achieved access to both the IVI (through the TCU) and the gateway, we could proceed to achieve our primary objectives - vehicle theft and remote control of vehicle functions.

So far, the described sequence of steps illustrates how weak authentication, default credentials, and lack of integrity checks on critical system components can compound into a compromise of the vehicle's internal network. In particular, the ability to move laterally between domains via the gateway highlights the systemic risk posed by inadequate hardening of Linux-based automotive systems.

5 Breaking In: Vehicle Theft

This objective was achieved by utilizing the vehicle's MQTT-based communication solution used for remote commands from the user application. MQTT (MQ Telemetry Transport) is a messaging protocol designed for devices with limited bandwidth or power, primarily used in IoT applications. In vehicles, MQTT facilitates communication between components such as the TCU, IVI, and external servers, while its publish-subscribe model also ensures scalability and efficiency for connected car systems. In the tested vehicle, an MQTT-based solution was developed to enable end-user interaction via a mobile application. This application allows users to perform various actions, such as unlocking the vehicle, starting it, turning on the AC, activating the headlights, and more.

When the end-user opens the mobile application, it wakes the vehicle via cellular communication and activates the MQTT server on the TCU modem. The application sends commands, such as unlocking the doors, to the cloud. The cloud processes the request and securely forwards (publishes) it to the TCU using an MQTT message over a TLS connection. Upon validation, it wakes up the CAN bus and its connected components. Next, it sends a CAN request to the BCM, which unlocks the doors. Finally, the TCU sends an acknowledgment to the cloud, confirming the action's success.

In earlier port scans on the IVI and gateway, we found that both had open ports typically associated with MQTT, suggesting they are likely subscribed to an MQTT server. Further investigation of the gateway's file system revealed a proprietary binary containing an application connected to the MQTT server. By reversing this binary, we extracted the IP address, username, password, and file system locations of the necessary certificates of the MQTT client, allowing us to subscribe to the MQTT server. It is also noteworthy that the MQTT server credentials were hardcoded in the TCU of each car and were identical across the entire fleet.

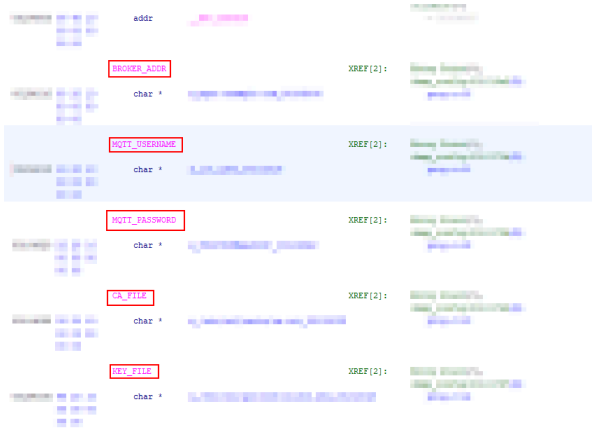

Figure 3. Reversing the binary reveals hardcoded MQTT credentials, including username, password, and paths to the client's key and certificates.

Once subscribed to the MQTT server, we sniffed and analyzed all published messages on the MQTT server. This analysis uncovered critical and private information, including the vehicle's location, VIN, current speed, navigation data, and CAN data from various sensors, such as lights and blower details. Additionally, we discovered base64-encoded messages containing cloud information, tokens, URLs, and other sensitive data. Using the gathered information and the control over the gateway, we replicated and executed all actions supported by the application, including unlocking the doors and starting the vehicle - the two operations needed to facilitate stealing the vehicle.

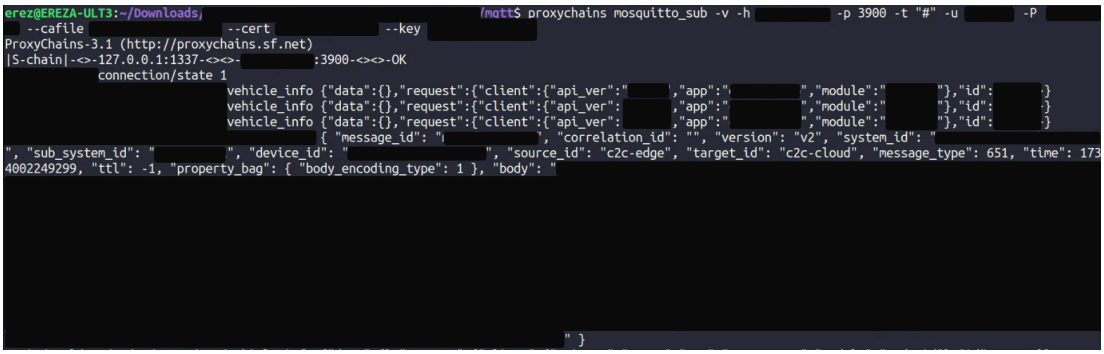

Figure 4. MQTT communication after subscribing to the secured channel using the stolen credentials

To summarize the attack scenario, we connected to the vehicle's WiFi hotspot, accessed the ADB interface, and established a persistent reverse shell (Figure 3, Steps 1-2). Using the remote connection, we tunneled communication to the gateway (Figure 3, Step 3), subscribed to the MQTT server (Figure 3, Step 4), and executed commands to unlock the doors and start the vehicle engine (Figure 3, Step 5).

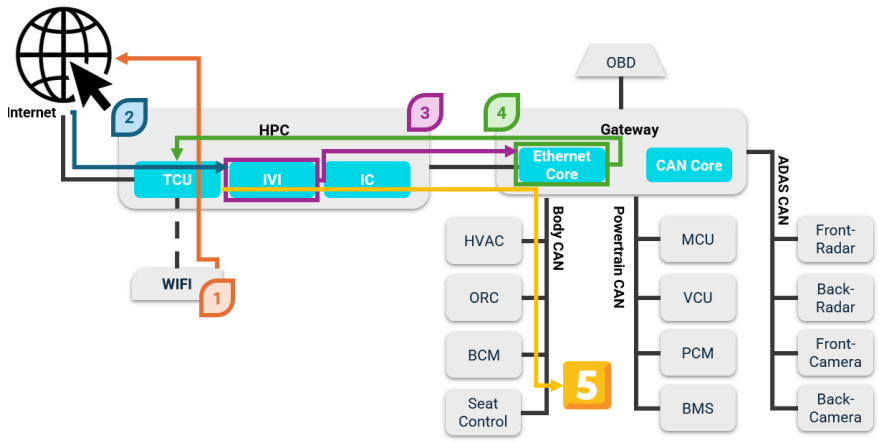

Figure 5. Partial vehicle-level architecture illustrating the theft attack scenario

The MQTT-based mechanism for remote commands had several flaws, including the use of a hardcoded and easily guessable password and username. The MQTT keys and certificates were stored in the file system, making them accessible to low-privileged users. Additionally, the same credentials were shared across the entire fleet. Using an HSM to securely manage keys and certificates would prevent unauthorized access. Finally, assigning unique passwords and keys to each vehicle would ensure that compromising one vehicle would not impact others, significantly enhancing overall fleet security.

6 Beyond Physical Access: Remote Compromise

After successfully stealing the vehicle, we moved on to our second objective: demonstrating how an attacker could cause physical damage or manipulate the behavior of safety-critical ECUs. We had already achieved remote shell access to the Linux core of the gateway and identified a persistent location where we could upload and execute our binaries. Moreover, the MQTT communication provided real-time information on the vehicles' location and speed to decide on the most impactful time to execute the attack.

The next step was to send CAN messages to relevant ECUs and to perform operations capable of causing physical damage. To perform commands on safety-critical ECUs, such as the BMS (Battery Management System) and the BCM (Body Control Module), it was necessary to access the CAN bus interfaces. We also needed a way to bypass the UDS security access mechanism to execute restricted operations like UDS ECU reset. While investigating the gateway's file system, we uncovered and reverse-engineered several proprietary binaries. We identified the presence of a UDS implementation in one of the binaries and subsequently focused our analysis on that file. Notably, the binaries contained debug symbols, which expedited the reverse-engineering process.

Initially, all relevant UDS services were protected behind the UDS security access mechanism (Service 0x27). This prevented us from reading, writing, or resetting any of the ECUs, necessitating a way to bypass this mechanism. During the reverse-engineering process, we identified a function that sends diagnostic request messages. Since symbols were left in compilation, the reversing process was simpler, enabling us to search for keywords such as 'key' and 'seed'. This search discovered that the function also calculated the key required as a response to a seed sent by the connected ECU. Reversing these functions allowed us to solve the Security Access for the vehicle, enabling restricted operations, such as ECU reset, on any ECU in the vehicle. The UDS security access mechanism was implemented within the system without adhering to cryptographic best practices. On top of that, testing during a driving test confirmed the attack's success, which highlighted another critical issue: safety-critical ECUs lacked a check on driving conditions on themselves.

CAN 1	000007B0	F	8	02 11 01 00 00 00 00 00
CAN 1	000007B8	FB	8	02 51 01 00 00 00 00 00

Figure 6. ECU-Reset success through the CAN bus

The final step was to identify a method for injecting CAN messages from the Ethernet core to the CAN core, which was then used to execute one of the newly available restricted methods. Further reverse-engineering of the binary revealed that the security access functions were invoked by a larger function performing an ECU Reset. Reversing this function uncovered the communication process between the cores. This process begins

with creating a TCP socket, followed by an attempt to connect to the DoIP server in the CAN core via TCP. In addition, the ECU reset function takes the Diagnostic ID of the target ECU as an argument. At that point, we had two options: develop a custom script to send the required CAN messages or modify the discovered binary.

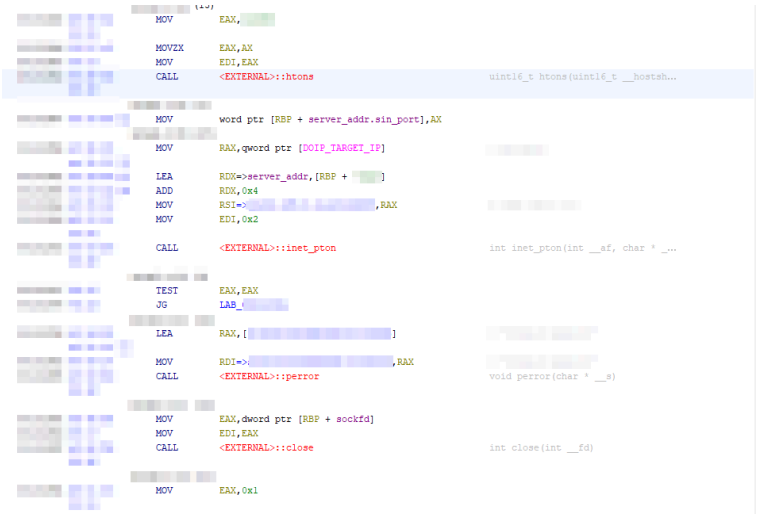

Figure 7. The reversed binary shows how a TCP socket is created to connect to the DoIP server in the CAN core.

Since time was of the essence, we decided to find a way to utilize the customer binary, creating an edited version capable of resetting a chosen ECU in an infinite loop. Further reverse engineering led us to the main function. We identified several threads initialized for running scheduled tasks. Specifically, we found that each thread executed a function that assumes a few specific conditions, calls a function that checks the status of the ECU, and, for a specific status, invokes the ECU reset function.

Figure 8. Calling structure of the functions

Looking at the Run() function, we observed a check on the ignition status. The Monitor() function was called only if the ignition was in a specific mode, and the ECUReset() function was executed only after additional checks, targeting a hard-coded ECU. This required modifying three parts of the binary: removing the ignition status check in Run(), eliminating all checks in Monitor(), and replacing the hardcoded ECU in ECUReset() with the relevant target ECU. Following these modifications, the update was completed successfully. We uploaded the modified binary to the gateway ECU via the SSH connection. Then, the binary

was tested in the vehicle, including in a driving scenario, showing successful resets of ECUs such as the BCM and VCU (Vehicle Control Unit). During these resets, the vehicle experienced sudden braking and shaking, even at low speeds, potentially posing a major risk to passengers if such incidents occurred while driving at higher speeds. Moreover, the tested car became non-operational and required a full battery reset to restore functionality.

Due to the provided time constraints, we focused on this specific use case. Given more time, developing a completely new binary capable of sending and receiving any diagnostic message to and from any ECU reachable through the gateway would have been possible.

In summary, we connected to the vehicle's WiFi hotspot, accessed the ADB interface, and established a persistent reverse shell (Figure 6, Steps 1-2). Using that remote connection, we tunneled communication to the gateway, subscribed to the MQTT server, and uploaded the modified binary (Figure 6, Step 3). When the MQTT traffic indicated that the vehicle was in a driving session, along with navigation data and speed, we executed the binary, triggering continuous ECU resets (Figure 6, Steps 4-5). As a result, the car braked and shook unexpectedly - endangering everyone inside - and eventually became non-operational until a battery reset was performed.

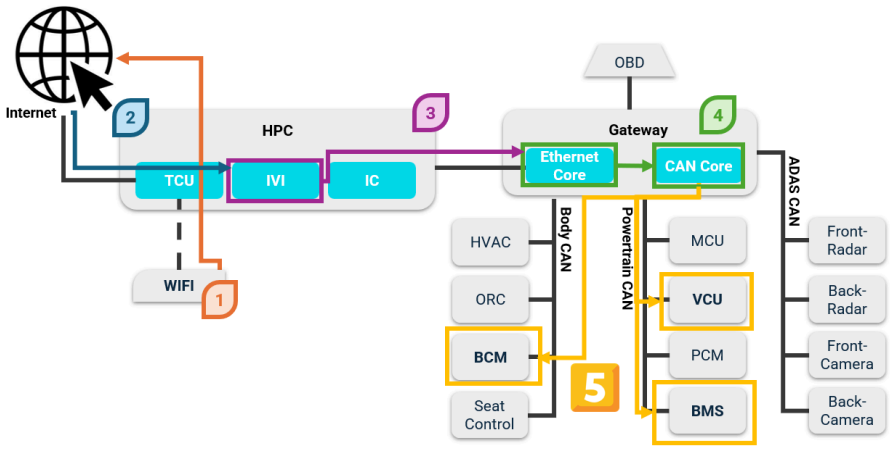

Figure 9. Partial vehicle-level architecture with the vehicle-compromising attack scenario.

7 Conclusions

The real-world case study presented above challenges a common belief in the automotive industry that causing significant damage requires extensive research time and effort. In just eight days, we demonstrated real-world attack scenarios on an EV just before SoP, exploiting common weaknesses such as unprotected debug interfaces, default credentials, lack of secure boot and proper integrity checks, hard-coded passwords, and a shaky UDS Security Access mechanism. This showed how an attacker could progress from minimal access to complete vehicle theft and compromise safety-critical systems, thus emphasizing the importance of building vehicles with properly implemented security mechanisms. Including the use of HSM, cryptographic algorithms, secure boot, and integrity checks to bolster vehicle security. Although some vulnerabilities may seem minor or uncommon, we encountered them in multiple projects and leveraged them in different attack scenarios. In addition, most of those issues are relatively easy to check and prevent. For example, checking the existence of a secure boot only requires creating a new file and restarting the system. The following table summarizes the main identified issues and their corresponding mitigations to guide secure vehicle development:

Issue	Mitigation
Unprotected debug interfaces	Disable or secure debug interfaces in production using authentication.
Use of default credentials	Enforce unique, strong credentials per device and disable default credentials before deployment.
Hard-coded passwords	Eliminate hard-coded passwords by using secure storage mechanisms like HSMs to manage secrets dynamically.
Lack of security boot and integrity checks	Implement secure boot and proper cryptographic integrity checks to verify firmware authenticity at startup.
Segregation (WiFi to IVI-ADB, SSH from the TCU to Gateway)	Enforce strict segregation between network domains using firewalls and role-based to isolate and control access to critical systems.
Debug symbols	Strip debug symbols from binaries to hinder reverse engineering efforts.
Shaky UDS Security Access Algorithm	Use a strong, non-predictable cryptographic challenge-response algorithm with an adequate key length.

Based on our findings, this paper offers two key takeaways. First, significant security issues in automotive systems do not always require prolonged research efforts; impactful

vulnerabilities can be identified within relatively short timeframes. Second, while Tier 1 suppliers and OEMs often focus on passing regulatory checks through isolated component-level ECU testing, this approach overlooks important system-level interactions. Certain attack surfaces and vulnerabilities only emerge when the vehicle is considered as a whole, with different ECUs functioning together as interconnected parts of a complex system. Comprehensive security assessments must therefore go beyond component testing to include full-vehicle analysis.

Moving forward, we hope that more stakeholders in the automotive industry will disclose such vulnerabilities and share insights on similar attacks. This shared knowledge will help us all learn from each other's mistakes and create a better and safer production environment for connected vehicles.

Acknowledgments

We thank Amarnath Kannadhasan, Giwansh Aryan, and Omer Mayraz for their valuable contributions to the project.