EV App VAPT Methodology

Target: Electric Vehicle App

Author: Barsanti Federico (CNR Pisa - IIT Intern)
Date: 2025-12-09

Contents

1. Executive Summary

1.1. Assessment Overview

In an era where vehicle security has migrated from metal keys to mobile applications, this document presents a systematic methodology for assessing the security of automotive carmaker applications: mobile apps that control physical vehicle systems through cloud-connected APIs. This methodology was developed and validated during a comprehensive security assessment of an automotive carmaker application for an electric vehicle platform, providing remote control capabilities for connected vehicles.

The assessment discovered **5 security vulnerabilities** affecting a platform with an estimated 500,000+ active users on android (Google Play Store Downloads, 2025), demonstrating how application-layer security flaws can enable unauthorized access to physical vehicle control systems. The findings illustrate a critical reality of automotive cybersecurity: **vulnerabilities in mobile authentication do not just compromise data, they compromise vehicles**.

1.2. Key Findings and Impact

For the complete, canonical list of findings and severities, see **§5.1 Vulnerability Findings Overview**. For the end-to-end attack chain and timeline, see **§5.2 Attack Chain Visualization**.

1.3. Methodology Highlights

1.3.1. Evidence-Grade Documentation

Institutional research requires higher evidence standards than typical security assessments. All findings backed by:

- **SHA-256 hashed evidence files:** Complete chain of custody for traffic captures and exploitation logs.
- **JSON Lines logging format:** Structured evidence with per-entry cryptographic integrity.
- **UTC timestamps:** Precise timeline reconstruction for all testing activities.
- **Reproducible test cases:** Step-by-step exploitation procedures validated in production environment.

1.3.2. Custom Tool Development

Manual testing limitations drove automation tool development:

1. `split_burp_xml.py` (138 lines Python): Split 51MB Burp Suite exports into AI-digestible 35-item chunks, enabling systematic analysis of 504 requests across 15 chunks.
2. `pin_bruteforce.py` (production-tested): Automated PIN testing with automatic token refresh, evidence-grade logging, ethical rate limiting.
3. `sanitize_for_github.py`: PII removal for portfolio publication while preserving technical structure.

These tools transformed infeasible manual processes (a significant amount of time to test all PINs) into practical automated testing (completed within a reasonable timeframe with 214 attempts validated).

1.4. Assessment Outcomes

1.4.1. Responsible Disclosure Approach

All identified vulnerabilities were reported following coordinated disclosure practices in accordance with OWASP and ISO/IEC 29147 standards. The disclosure process involved internal institutional review, coordination with legal advisors, and vendor notification through appropriate security channels. Following industry-standard 90-day disclosure timelines (adjusted for automotive patch development complexity), adequate time was provided for vendor validation and remediation before public documentation. Disclosure timing balanced automotive industry patch development requirements against the need for transparency in security research, with final publication authorized per CNR institutional policies.

1.5. Methodology Value Proposition

1.5.1. For Security Professionals

- Repeatable assessment framework applicable to any automotive carmaker application
- AI-assisted analysis techniques with validation protocols to prevent hallucinations
- Custom tool development patterns for scaling manual testing
- Evidence documentation standards meeting institutional research requirements

1.5.2. For Automotive Manufacturers

- Compliance-mapped findings aligned with UNECE R155 and ISO/SAE 21434 requirements
- Real-world attack chain demonstrations showing exploitation feasibility
- Remediation guidance with code examples and architectural recommendations
- Threat modeling approach considering complete mobile → cloud → vehicle attack surface

—

2. Introduction

2.1. From Physical Keys to API Endpoints

For over a century, vehicle security relied on a simple principle: physical possession. A door lock required a metal key in your hand, shaped to match pins inside a mechanical cylinder. An ignition system demanded that same key turned in a steering column switch. This security model was intuitive: if someone wanted to steal your car, they needed to be physically present with a tool to defeat mechanical barriers.

That world is ending.

When you unlock your vehicle from a mobile app today, you are executing a command that bypasses this entire physical security paradigm. A door lock that once required tactile manipulation now responds to an HTTP PUT request. An immobilizer that verified metal key patterns now validates JWT tokens. The fundamental shift from physical access control to digital authorization creates unprecedented attack surfaces where authentication flaws do not just leak data, **they enable unauthorized control of 2-ton machines moving at highway speeds**.

Consider the attack chain: A mobile application authenticates via OAuth2, receives a refresh token valid for an extended period, sends that token to a cloud API gateway, which forwards commands through a cellular connection to the vehicle's Telematics Control Unit (TCU), which translates HTTP requests into CAN bus messages, which finally trigger physical actuators that mechanically unlock doors or disable alarms. This is a seven-layer system where a vulnerability in layer one (mobile app authentication) propagates consequences all the way to layer seven (physical vehicle state).

Critical Prerequisite: However, this attack chain assumes successful compromise of user credentials (email + password), typically achieved through credential stuffing (0.1-2% success rate from breach databases), targeted phishing campaigns (10-30% success rate), or social engineering. Password acquisition represents the primary attack barrier, often requiring weeks to months of reconnaissance and multiple compromise attempts with significant detection risk. The authentication vulnerabilities documented in this assessment (weak PIN brute-force protection, missing timestamp validation) become exploitable only AFTER this credential compromise phase succeeds. Multi-Factor Authentication (MFA) on sign-in would completely break this attack chain by preventing password-only authentication, regardless of subsequent authentication weaknesses.

2.2. The Automotive Cybersecurity Reality

Modern vehicles are no longer isolated mechanical systems, they are interconnected cyber-physical platforms with attack surfaces spanning mobile applications, cloud infrastructure, cellular networks, and embedded vehicle systems [1], [2]. The 2015 Jeep Cherokee hack demonstrated this reality when researchers remotely disabled the vehicle's transmission on a highway by exploiting vulnerabilities in the entertainment system's cellular connection [3]. Tesla has addressed multiple vulnerabilities in their mobile apps and vehicle ecosystem through their public bug bounty program, including authentication weaknesses enabling unauthorized vehicle access [4].

These are not theoretical concerns. Industry projections indicate that global production is approaching 100 million connected vehicles annually as telematics integration becomes standard across new models [5].

Each modern vehicle typically contains:

- **100 Electronic Control Units (ECUs)** managing functions from infotainment to brake control [6]
- **Tens to hundreds of millions of lines of code**, reflecting software complexity exceeding many aerospace systems [7]
- **4G/5G cellular connectivity** enabling continuous telematics, telemetry upload, remote control, and OTA updates [6]
- **Mobile carmaker apps (iOS/Android)** capable of issuing remote control commands such as lock/unlock, climate activation, or locating the vehicle [1]
- **Cloud backend infrastructure** using API gateways, authentication stacks, and command-processing pipelines to mediate driver-to-vehicle communication [7]

Every connection point represents a potential entry vector. Every authentication mechanism becomes a security boundary operating under adversarial pressure.
Every remote command issued through these systems is a digital instruction that will physically manipulate mechanical components affecting human safety.

2.3. Why Automotive Application Security Is Different

Unlike traditional mobile application security testing that focuses on data confidentiality and integrity, **automotive application security directly impacts physical safety**. The stakes are fundamentally different:

Traditional Application Security Stakes	Automotive Application Security Stakes
Authentication bypass ↳ Unauthorized data access	Authentication bypass ↳ Unauthorized vehicle control
Session hijacking ↳ Account compromise	Session hijacking ↳ Remote door unlock capability
API vulnerability ↳ Information disclosure	API vulnerability ↳ Command injection to vehicle systems
Data breach ↳ Privacy violation	Data breach ↳ Real-time location tracking + physical access

When an authentication flaw in a banking app is exploited, the damage is financial and can be reversed through fraud protection systems. When an authentication flaw in a vehicle control app is exploited, the attacker can physically unlock the vehicle, disable the alarm, and potentially start the engine, **all without cryptographic or mechanical barriers to prevent these actions**.

This methodology addresses this unique threat landscape by:

1. **Prioritizing control plane security over data plane:** Vehicle command authorization takes precedence over data privacy.
2. **Mapping digital vulnerabilities to physical impacts:** Every finding assessed for real-world exploitation scenarios.
3. **Considering complete attack chains:** From mobile app compromise to CAN bus command execution.
4. **Evaluating Android OS security context:** Realistic threat modeling considering platform protections.
5. **Applying automotive cybersecurity standards:** Compliance with UNECE R155, ISO/SAE 21434.

2.4. Project Context: Automotive Carmaker Application Assessment

This methodology was developed and validated during a comprehensive security assessment of an automotive carmaker application for an electric vehicle platform, conducted as part of a research internship at CNR (Consiglio Nazionale delle Ricerche) - Istituto di Informatica e Telematica (IIT) in Italy. The assessment scope encompassed:

2.4.1. Target Platform: Automotive Carmaker Application

- **Purpose:** Remote vehicle control, monitoring, and management for electric vehicle.
- **Functionality:** Door lock/unlock, engine start/stop, climate control, location tracking, vehicle diagnostics.
- **User base:** 500,000+ active users (estimated).
- **Physical impact:** Commands directly control vehicle actuators via CAN bus.

2.4.2. Assessment Scope

```
Mobile Application (Android)
    ↓ HTTPS/TLS
Cloud API Gateway (manufacturer API endpoint)
    ↓ Proprietary Protocol
Vehicle Telematics Control Unit (TCU)
    ↓ CAN Bus Messages
Physical Actuators (Door Locks, Immobilizer, Climate Control)
```

2.4.3. Key Findings

See **§5.1 Vulnerability Findings Overview** for the complete table of findings. See **§5.2 Attack Chain Visualization** for the attack chain and timeline.

2.4.4. Authorization Context

- Conducted under CNR institutional research authorization
- Used dedicated test accounts created specifically for security assessment
- Followed responsible disclosure practices (coordinated with vendor before public release)
- All testing performed in controlled environment respecting production system integrity

2.5. Methodology Philosophy

This methodology is evidence-driven (reproducible traffic, SHA-256 hashes, UTC timestamps, full chain-of-custody) and uses AI to accelerate pattern discovery **with mandatory human validation**. For the complete workflow and validation protocol, see **§4.5.3 AI-Augmented Analysis with Zero-Hallucination Validation**.

3. Background

This chapter consolidates all standards, technical foundations, and tools used in the assessment. By introducing these concepts once comprehensively, later chapters can reference them without repeating explanations.

3.1. Automotive Cybersecurity Standards and Regulations

The automotive industry recognizes cybersecurity threats and regulatory bodies worldwide now mandate cybersecurity engineering practices:

3.1.1. UNECE R155 (United Nations Economic Commission for Europe Regulation No. 155)

UN Regulation No. 155 is a European Economic Commission regulation that entered into force in January 2021, establishing legally binding cybersecurity requirements for vehicles sold in the European Union, Japan, and South Korea. This regulation mandates that vehicle manufacturers implement formal cybersecurity management systems covering the entire vehicle lifecycle—from initial design through production and post-market maintenance. For connected vehicle applications like mobile carmaker apps, R155 requires manufacturers to demonstrate:

- Systematic threat analysis and risk assessment (TARA) for all external connectivity points.
- Secure software development processes with vulnerability management protocols.
- Monitoring and response procedures for cybersecurity incidents throughout the vehicle's operational life.
- Software update mechanisms with integrity validation to prevent unauthorized modifications.

The regulation applies not only to embedded vehicle systems but also to mobile applications and cloud services that enable vehicle control functions, making it directly relevant to this assessment.

3.1.2. UNECE R156 (Software Update Regulation)

UN Regulation No. 156 complements UNECE R155 by establishing mandatory cybersecurity requirements specifically for **software updates** in road vehicles. Adopted in 2021, it ensures that all vehicles and manufacturers implement robust **Software Update Management Systems (SUMS)** to maintain cybersecurity integrity throughout a vehicle's lifetime.

The regulation mandates that manufacturers:

- Implement processes to securely **deliver, authenticate, and verify** software updates—both over-the-air (OTA) and via physical media.
- Maintain **traceability and integrity validation** of all software components, preventing unauthorized or malicious code introduction.
- Ensure that update packages are **cryptographically signed** and validated before installation.
- Provide mechanisms for **rollback or recovery** in case of update failure, preserving vehicle safety.
- Retain auditable records demonstrating that updates do not adversely affect vehicle functionality or safety systems.

Relevance to Mobile Application Testing: For connected vehicle ecosystems, compliance with R156 extends beyond embedded ECUs to include **mobile applications and cloud services** participating in the update process. Applications responsible for initiating or authorizing OTA updates must:

- Use **authenticated, integrity-checked communication** with backend servers.
- Validate update metadata before invoking vehicle update procedures.
- Prevent unauthorized triggering of software update operations through authentication and replay-resistant command flows.

This regulation bridges the operational gap between cybersecurity management (R155) and software lifecycle assurance (R156), ensuring that secure update delivery remains an active, verifiable process over the entire vehicle lifespan.

3.1.3. ISO/SAE 21434 (Road Vehicles and Cybersecurity Engineering)

ISO/SAE 21434:2021 provides the engineering framework and technical methodologies for implementing UNECE R155 requirements. Published jointly by the International Organization for Standardization (ISO) and the Society of Automotive Engineers (SAE), this standard defines systematic processes for integrating cybersecurity into automotive development lifecycles. Key provisions include:

- Risk-based cybersecurity engineering approach tailored to automotive cyber-physical systems.
- Threat analysis and risk assessment (TARA) methodologies considering attacker capabilities and attack vectors.
- Cybersecurity validation requirements ensuring security measures function as intended.
- Vulnerability management processes for addressing security issues discovered post-release.

This standard emphasizes that mobile applications controlling vehicle functions must undergo the same rigorous security assessment as embedded vehicle systems due to their potential impact on vehicle safety and security.

3.1.4. NIST Cybersecurity Framework (Applied to Automotive Context)

The **NIST Cybersecurity Framework**, developed by the U.S. National Institute of Standards and Technology, provides a voluntary risk-management approach applicable across industries, including automotive. While not automotive-specific like ISO/SAE 21434, the framework's five core functions offer complementary guidance for securing connected vehicle ecosystems:

- **Identify:** Asset inventory, risk assessment, governance. (e.g., cataloging all mobile app endpoints that trigger vehicle commands)

- **Protect:** Access control, data security, protective technology. (e.g., authentication mechanisms, session management, input validation)

- **Detect:** Anomaly detection, security monitoring, intrusion detection. (e.g., detecting unusual API access patterns or unauthorized control attempts)

- **Respond:** Incident response planning, communication, analysis. (e.g., procedures for addressing compromised accounts or exploited vulnerabilities)

- **Recover:** Recovery planning, improvements, lessons learned. (e.g., restoring normal operations after security incidents, implementing preventive controls)

The framework's flexibility allows integration with automotive-specific standards (UNECE R155, ISO/SAE 21434) while providing structured guidance for organizational cybersecurity maturity assessment.

3.1.5. OWASP Mobile Top 10 (2024)

The **OWASP Mobile Security Project** provides industry-standard guidance for mobile application security. The Mobile Top 10 (2024 edition) identifies the most critical mobile security risks:

- **M1: Improper Credential Usage** - Hardcoded API keys, passwords, or cryptographic keys in application code.
- **M2: Inadequate Supply Chain Security** - Compromised third-party libraries or SDKs.
- **M3: Insecure Authentication/Authorization** - Weak authentication mechanisms or broken authorization logic.
- **M4: Insufficient Input/Output Validation** - Lack of validation enabling injection attacks.
- **M5: Insecure Communication** - Unencrypted data transmission or weak TLS configurations.
- **M6: Inadequate Privacy Controls** - Excessive permissions or PII exposure.
- **M7: Insufficient Binary Protections** - Lack of code obfuscation or anti-tampering measures.
- **M8: Security Misconfiguration** - Improper platform or server configuration.
- **M9: Insecure Data Storage** - Sensitive data stored without encryption.
- **M10: Insufficient Cryptography** - Weak algorithms or improper cryptographic implementations.

The assessment findings map directly to M1 (hardcoded client secret), M3 (weak PIN brute-force protection), and M4 (missing stamp validation).

3.2. Technical Architecture and Foundations

3.2.1. OAuth2 Background: Mobile App Authentication Fundamentals

Note: This subsection provides OAuth2 background research conducted during the internship to understand the authentication mechanisms used in automotive mobile applications. Understanding these fundamentals was essential for analyzing the traffic patterns and identifying authentication-related vulnerabilities documented in Phase 4.

3.2.1.1. What is OAuth2?

OAuth 2.0 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service. Rather than sharing passwords directly with third-party applications, OAuth2 allows users to grant access through a token-based system. In the automotive context, mobile carmaker apps use OAuth2 to securely communicate with vehicle control APIs—the app authenticates once, receives tokens, and uses those tokens to authorize commands like remote door unlock or climate control activation.

The framework involves three key components: (1) the **Authorization Server** (issues tokens after validating credentials), (2) the **Resource Server** (the vehicle control API that validates tokens and executes commands), and (3) the **Client** (the mobile application). When you log into an automotive carmaker app, you are interacting with an OAuth2 Authorization Server that issues access tokens permitting your app to control your vehicle.

3.2.1.2. Client Types: The Critical Security Distinction

OAuth2 defines two types of clients based on their ability to maintain credential confidentiality:

Confidential Clients are applications running in secure environments—typically backend servers—that can safely store secret credentials. A web server application can keep a `client_secret` in its configuration files protected by server access controls and never exposed to end users. These clients can safely use client secrets as part of the authentication flow because the credentials remain on trusted infrastructure.

Public Clients are applications that cannot securely store credentials—this includes mobile applications (Android/iOS), single-page applications (SPAs), and any software distributed to end users. Mobile apps are distributed as APK or IPA files that users install on their devices. These files can be decompiled, reverse-engineered, and analyzed, meaning any “secret” embedded in the application code is effectively public information. An attacker with the APK file can extract embedded credentials in minutes using tools like JADX or apktool.

The Mobile App Problem: Despite being public clients, many production mobile applications embed OAuth2 client secrets as if they were confidential clients. This represents a fundamental misunderstanding of the OAuth2 security model. When automotive apps follow this anti-pattern—as observed in APP-F-001 during this assessment—they create a vulnerability where APK decompilation exposes credentials enabling token request forgery.

3.2.1.3. OAuth2 Tokens in Practice

OAuth2 uses two primary token types with different lifecycles and purposes:

Access Tokens are short-lived credentials (typically 30-60 minutes validity) that authorize specific API requests. When the mobile app needs to send a command to the vehicle—such as "unlock doors"—it includes the access token in the HTTP Authorization header. The vehicle control API validates this token and, if valid and unexpired, executes the requested command. Short token lifespans limit the attack window if a token is intercepted or stolen.

Refresh Tokens are long-lived credentials (days to weeks) used to obtain new access tokens without requiring the user to re-authenticate. When an access token expires, the app uses the refresh token to request a new access token from the Authorization Server. During the assessment, refresh tokens were observed with 30-day validity periods—meaning a stolen refresh token grants an attacker 30 days of persistent access capability. Refresh tokens should be rotated (a new refresh token issued with each use) to limit compromise windows, but rotation was not observed during this assessment.

3.2.1.4. Why Client Secrets Do Not Belong in Mobile Apps

RFC 8252 ("OAuth 2.0 for Native Apps") explicitly recommends that mobile applications **should not use client secrets** because they cannot be kept confidential. Instead, the standard recommends using **Proof Key for Code Exchange (PKCE)**, which provides security without relying on pre-shared secrets that would be exposed through APK decompilation [8].

When a mobile app embeds a client secret (as found in APP-F-001: `client_secret: "secret"`), the security implications are:

1. **Credential Exposure**: Any attacker can decompile the APK and extract the client secret in plaintext.
2. **Token Request Forgery**: With the client credentials, an attacker can impersonate the legitimate application and request tokens.
3. **No Remediation Path**: Once the client secret is public, it cannot be "un-published"—changing it breaks all existing app installations.

The assessed application's vulnerability APP-F-001 demonstrates this exact scenario: the client secret was hardcoded in the APK, discoverable through JADX decompilation, enabling token request forgery without requiring the legitimate application. While the client secret alone cannot compromise user accounts (valid user credentials are still required), it undermines the OAuth2 security model by enabling client impersonation.

Further Reading:
- RFC 6749: The OAuth 2.0 Authorization Framework
- RFC 8252: OAuth 2.0 for Native Apps. (mobile best practices)
- OWASP Mobile Top 10 - M1: Improper Credential Usage

3.2.2. Android 7+ Certificate Trust Model

3.2.2.1. The Pre-Android 7 Era: Traditional Certificate Trust Model

Before Android 7.0 (API 24, released August 2016), mobile application security testing followed a straightforward certificate configuration workflow:

1. Install Burp Suite CA certificate via `Settings → Security → Install from storage`
2. Configure device proxy to `192.168.1.100:8080`
3. Open app → Burp captures all HTTPS traffic

Apps trusted user-installed certificates by default. This made sense for user-initiated browser traffic (user installs bank's CA to access internal portal) but created a security vulnerability: any app could intercept any other app's HTTPS traffic by convincing the user to install a malicious CA certificate.

3.2.2.2. Android 7 Split Trust Model (The Breaking Change)

Android 7 introduced **split certificate stores** to solve this vulnerability:

- **User Certificate Store** (`/data/misc/user/*/cacerts-added/`): Certificates installed by user via Settings. Trusted for **browser** traffic only.
- **System Certificate Store** (`/system/etc/security/cacerts/`): Certificates bundled with Android OS. Trusted for **all** app traffic unless app opts out.

Apps targeting API 24+ (Android 7+) **ignore user-installed certificates by default**. From the app's perspective, Burp's CA certificate might as well not exist. The proxy sees encrypted TLS handshakes but cannot decrypt application-layer HTTP.

Exception - Network Security Configuration: Apps can explicitly opt-in to trust user certificates by declaring a Network Security Configuration in their AndroidManifest.xml, though this is rare in production automotive applications due to security implications.

Impact on security testing: The traditional workflow fails silently. Burp shows HTTPS connections but displays "TLS: encrypted_handshake" instead of "GET /api/v1/user/profile". Without understanding this change, testers might conclude "the app has no interesting API traffic" when in reality they simply cannot see it.

3.2.2.3. Why System-Level Certificate Installation is Required

The solution requires placing Burp's CA certificate in the **system** trust store where apps targeting API 24+ will honor it. This is only possible with **root access**, the system partition is read-only on non-rooted devices by design (preventing malware from installing system-trusted certificates).

3.3. Assessment Tools and Technologies

This section introduces all tools used throughout the assessment. Later chapters reference these tools without repeating technical details.

3.3.1. Burp Suite Professional

Purpose: HTTPS traffic interception, manipulation, and analysis

Version Used: 2023.10.3.7

Key Features:
- **Proxy**: Intercept and modify HTTP/HTTPS requests and responses in real-time.
- **Repeater**: Manually craft and resend individual requests for testing.
- **Intruder**: Automated payload fuzzing and brute-force testing.
- **Decoder**: Base64, URL, HTML encoding/decoding utilities.
- **Comparer**: Diff tool for analyzing request/response variations.

Configuration Requirements:
- Proxy listener on `0.0.0.0:8080` (accessible from emulator/device)
- CA certificate exported and installed as system-trusted on test devices.
- HTTP history filters configured to capture only target application traffic.

Assessment Usage: Primary tool for HTTPS traffic capture and analysis throughout all phases. See **§3.3.1 Burp Suite Professional** for detailed Burp Suite configuration procedures.

3.3.2. Frida Dynamic Instrumentation Framework

Purpose: Runtime code manipulation and SSL pinning bypass

Version Used: 16.0.19 (android-x86_64)

Key Capabilities:
- **Dynamic Hooking**: Intercept and modify function calls at runtime without app recompilation.
- **SSL Pinning Bypass**: Override certificate validation functions to enable HTTPS interception.
- **Memory Inspection**: Read/write application memory for behavior analysis.
- **JavaScript API**: Write instrumentation scripts in JavaScript for rapid prototyping.

Core Components:
- `frida-server`: Background service running on device with root privileges.
- `frida-tools`: CLI tools (`frida`, `frida-ps`, `frida-trace`) for interaction.
- **CodeShare Scripts**: Community-contributed SSL pinning bypass scripts. (e.g., `universal-android-ssl-pinning-bypass-with-frida`)

Assessment Usage: Essential for defeating SSL certificate pinning, enabling HTTPS traffic capture. See **§4.4 Phase 3: SSL Pinning Bypass**.

3.3.3. JADX (Dex to Java Decompiler)

Purpose: APK decompilation and static code analysis

Version Used: 1.4.7

Key Features:
- **DEX to Java**: Converts Android's Dalvik bytecode to readable Java source code.
- **Resource Extraction**: Accesses AndroidManifest.xml, strings.xml, and other app resources.
- **Search Functionality**: Full-text search across decompiled codebase.
- **Cross-References**: Navigate from method calls to implementations.

Assessment Usage: Static analysis for confirming dynamic findings, discovering hardcoded credentials, and understanding authentication logic. See **§4.6 Phase 5: Static Code Analysis** for static analysis methodology.

3.3.4. Android Debug Bridge (adb)

Purpose: Device management and debugging

Key Commands:

```bash
adb remount                  ⇒ Remount system partition as read-write
adb push file /path          ⇒ Copy file to device
adb shell                    ⇒ Interactive shell on device
adb install app.apk          ⇒ Install application
adb logcat                   ⇒ View system logs
```

Assessment Usage: Device configuration, certificate installation, app management, log monitoring.

3.3.5. Custom Python Analysis Tools

3.3.5.1. split_burp_xml.py

Purpose: Split large Burp Suite XML exports into manageable chunks for AI-assisted analysis

Functionality: Divides XML files into 35-item chunks, enabling systematic analysis of large traffic captures (e.g., 504 authentication requests split into 15 analyzable chunks)

Usage: See **§4.5.2 XML Splitting for AI-Assisted Analysis** Chapter 3, Section 3.4 for traffic analysis methodology

3.3.5.2. pin_bruteforce.py

Purpose: Automated PIN brute-force testing with evidence-grade logging

Features:
- Automatic bearer token refresh when expired
- JSON Lines logging format with SHA-256 integrity
- Ethical rate limiting respecting API constraints
- Configurable PIN ranges (common PINs, sequential, full keyspace)

Usage: See **§5.3.5 PIN Brute-Force (APP-F-002)** for vulnerability validation

3.3.5.3. sanitize_for_github.py

Purpose: PII removal for portfolio publication while preserving technical structure

Redactions: VINs, email addresses, device IDs, bearer tokens, refresh tokens

3.3.6. Wireshark

Purpose: Supplementary packet-level analysis

Version Used: 4.0.6

Assessment Usage: Frame-level correlation with Burp Suite captures, TLS handshake analysis, timing validation

3.3.7. Android Studio AVD Manager

Purpose: Android emulator creation and management

Version Used: 2023.1.1.26

Key Configuration:
- Device Profile: Pixel 4
- System Image: Google APIs x86_64 (rootable)
- Android Version: 16.0 ('Baklava')
- RAM: 4GB, Storage: 8GB

Assessment Usage: Primary testing environment for rapid iteration. **§4.2 Phase 1: Environment Setup** for emulator configuration details.

4. Methodology

This chapter presents the 9-phase assessment workflow with detailed implementation guidance. Phases 1 and 4 receive expanded coverage as they represent the foundation and core of the methodology.

4.1. Assessment Architecture

Automotive carmaker app security assessment requires evaluating three interconnected layers:

```
Layer 1: Mobile Application
├─ Static Analysis: APK decompilation, code review, secret scanning
├─ Dynamic Analysis: Traffic capture, SSL bypass, behavior monitoring
└─ Security Controls: Authentication flows, token management, root detection

        ↓ HTTPS/TLS Communication

Layer 2: Cloud Backend
├─ API Security: Authentication endpoints, authorization logic, rate limiting
├─ Session Management: Token lifecycle, refresh patterns, expiration handling
└─ Input Validation: Command injection, parameter tampering, stamp validation

        ↓ Telematics Protocol

Layer 3: Vehicle Systems (Out of Scope for Mobile-Focused Assessment)
├─ Telematics Control Unit (TCU): Command translation, CAN bus interface
├─ CAN Bus Network: ECU communication, actuator control
└─ Physical Actuators: Door locks, immobilizer, climate control
```

Assessment Focus: This methodology concentrates on Layers 1 and 2 (mobile app and cloud API), evaluating how vulnerabilities in these layers can propagate to Layer 3 (physical vehicle control). While vehicle embedded systems are out of scope, the methodology assesses whether mobile/cloud vulnerabilities can enable unauthorized commands that the vehicle will execute.

4.2. Phase 1: Environment Setup (Detailed)

Purpose: Establish testing infrastructure with HTTPS traffic interception capabilities

Why First: Every subsequent phase depends on the ability to capture and analyze HTTPS traffic. Without properly configured environment, vulnerability discovery is impossible.

4.2.1. Assessment Prerequisites

Hardware:
- Development workstation (Linux Based Distribution, 16GB RAM, x86_64 or ARM64 CPU)
- Physical Android test device (rooted) - for validation
- USB debugging cables

Software Tools:
- Android Studio (AVD Manager for emulator)
- Burp Suite Professional
- Frida (dynamic instrumentation framework)
- JADX (APK decompiler)
- Python 3.10+ (for analysis scripts)
- OpenSSL (certificate operations)

Knowledge Prerequisites:
- Android security model fundamentals (certificate trust, app sandboxing)
- HTTP/HTTPS protocol understanding
- Basic OAuth2 flow knowledge
- Command-line proficiency (bash, adb)

4.2.2. Dual-Device Strategy: Speed vs Validation

Android Emulator (Primary):
- **Purpose**: Fast iteration, snapshot/restore, root access by default
- **Configuration**: Pixel 4, Google APIs x86_64 (rootable), Android 16.0
- **Advantage**: Rapid snapshot/restore enabling iterative testing (10-second reset vs physical device reboot times)

Physical Device (Validation):
- **Purpose**: Hardware sensors, manufacturer ROM, real-world conditions
- **Configuration**: Samsung Galaxy J7 (SM-J730F), Android 9.0, TWRP + Magisk root
- **Advantage**: Anti-emulator check validation, real-world exploit confirmation

Critical Insight: 3 of 5 vulnerabilities required physical device validation to confirm real-world applicability beyond emulator artifacts.

4.2.3. Android Certificate Trust Configuration

See **§3.2.2 Android 7+ Certificate Trust Model** for complete Android 7+ certificate trust model background.

Solution Implemented: System-level certificate installation requiring root access

Emulator Configuration:

```bash
# Launch emulator with writable system partition
emulator -avd Pixel_4 -writable-system -no-snapshot -verbose &

# Enable root access
adb root

# Remount system partition as read-write
adb remount

# Convert Burp CA to Android system format
openssl x509 -inform DER -in burp-ca.der -out burp-ca.pem
CERT_HASH=$(openssl x509 -inform PEM -subject_hash_old -in burp-ca.pem | head -1)
cp burp-ca.pem ${CERT_HASH}.0

# Install in system trust store
adb push ${CERT_HASH}.0 /system/etc/security/cacerts/
adb shell chmod 644 /system/etc/security/cacerts/${CERT_HASH}.0
adb reboot

# Create baseline snapshot after configuration
adb emu avd snapshot save baseline_configured
```

Physical Device Configuration:
1. Unlock bootloader: `fastboot oem unlock`
2. Flash TWRP recovery: `fastboot flash recovery twrp.img`
3. Install Magisk for systemless root
4. Install certificate in `/system/etc/security/cacerts/` via TWRP

4.2.4. Burp Suite Configuration

See **§3.3.1 Burp Suite Professional** for complete Burp Suite tool introduction.

Proxy Listener Setup:
- Address: `0.0.0.0` (all interfaces)
- Port: `8080`
- TLS: Generate CA certificate, export for device installation

Device Proxy Configuration:

```bash
# Emulator (host machine is 10.0.2.2 from emulator)
adb shell settings put global http_proxy 10.0.2.2:8080

# Physical device (replace with workstation IP)
adb shell settings put global http_proxy 192.168.1.100:8080

# Verify connectivity
adb shell curl http://burp
# Should display Burp welcome page
```

HTTP History Filters:
- Filter by host: Only target application API domains
- Hide images, CSS, JavaScript: Focus on API traffic
- Show only in-scope items: Reduce noise from analytics

4.2.5. Validation and Troubleshooting

Environment Validation Checklist:

☐ Emulator launches with writable system partition
☐ Root access confirmed (`adb root` succeeds)
☐ Burp CA certificate installed in system trust store
☐ Device proxy configured pointing to Burp Suite
☐ HTTPS traffic decrypted in Burp Suite HTTP history
☐ Baseline snapshot created for rapid restoration

Common Issues:
- **"Certificate not trusted"**: Verify certificate in `/system/etc/security/cacerts/` with correct permissions (644)
- **"No traffic in Burp"**: Check proxy configuration with `adb shell settings get global http_proxy`
- **"Encrypted traffic only"**: SSL pinning active → Proceed to Phase 3 (Frida bypass)

4.2.6. Outcome: Foundation Enabled

This phase enables:
- Full HTTPS traffic inspection for all app features
- Rapid iteration through emulator snapshot/restore
- Real-world validation on physical hardware
- Evidence capture with cryptographic integrity (SHA-256 hashes)

Without this foundation: Subsequent phases cannot proceed—SSL pinning bypass requires traffic observation, vulnerability discovery requires request analysis, exploitation requires request manipulation.

4.3. Phase 2: Reconnaissance and Enumeration

Purpose: Map application features, discover API endpoints, understand functionality

Automotive Focus: Prioritize vehicle control operations (remote unlock, start, climate) over data retrieval

Deliverables:
- API endpoint catalog (27 endpoints identified across 5 feature categories)
- Feature matrix mapping user actions to HTTP requests
- Authentication flow diagram showing token types and lifecycles

Key Activities:
1. Manual app exploration with Burp Suite traffic capture
2. API endpoint extraction from decompiled APK (JADX)
3. Feature categorization (Authentication, Pairing, Control, Monitoring, Notifications)
4. Priority ranking based on security impact (vehicle control > monitoring > analytics)

4.4. Phase 3: SSL Pinning Bypass

Purpose: Defeat certificate pinning to enable HTTPS traffic inspection

See **§3.3.2 Frida Dynamic Instrumentation Framework** for complete Frida tool introduction.

Challenge: Applications implement certificate pinning to prevent MITM attacks, blocking Burp Suite interception even with system-trusted certificates.

Solution: Frida dynamic instrumentation with community SSL bypass scripts

Implementation:

Bash

```bash
# Start Frida server on device
adb push frida-server-16.0.19-android-x86_64 /data/local/tmp/frida-server
adb shell "chmod 755 /data/local/tmp/frida-server"
adb shell "/data/local/tmp/frida-server &"

# Run universal SSL pinning bypass script
frida -U -f com.manufacturer.app --codeshare pcipolloni/universal-android-ssl-
pinning-bypass-with-frida --no-pause
```

Validation: Burp Suite HTTP history shows decrypted HTTPS traffic for all app features including vehicle control commands.

4.5. Phase 4: Traffic Capture and Analysis (Detailed)

Purpose: Systematically capture and analyze HTTP/HTTPS traffic for each feature to discover vulnerabilities

Why Extended: Most vulnerability discovery happens through traffic pattern recognition

4.5.1. The Scale Problem

A single user flow (sign-in) generates **504 requests**, including:
- Authentication handshakes (OAuth2 token exchange)
- Device registration and fingerprinting
- Analytics telemetry (30+ tracking endpoints)
- Image asset fetches
- Third-party SDK communication
- Background sync operations

Challenge: Of 504 requests, perhaps 50 are security-relevant—finding signal in noise at scale.

4.5.2. XML Splitting for AI-Assisted Analysis

Problem: Burp Suite exports 51MB XML (Base64-encoded) exceeding AI context limits

Solution: Split into 35-item chunks preserving multi-request patterns

Why 35 Items:
- Too small (25 items): Breaks authentication flow patterns spanning 10+ requests
- Too large (100 items): Exceeds AI context window, degrades analysis quality
- Goldilocks (35 items): 3.4MB chunks preserving 95% of multi-request patterns

See **§3.3.5 Custom Python Analysis Tools** for `split_burp_xml.py` tool introduction.

Usage:

```bash
python3 split_burp_xml.py signin_http_history.xml
# Output: 15 chunks for 504 requests, each 35 items
```

4.5.3. AI-Augmented Analysis with Zero-Hallucination Validation

AI Role (Claude Code): Pattern acceleration across large datasets
- Search 35-item chunks for specified patterns (tokens, credentials, rate limiting)
- Extract matching requests with context
- Flag potential security issues for human validation

Human Role (Security Researcher): Validation and assessment
- Validate every AI claim against original XML
- Reproduce findings in Burp Repeater
- Confirm through static analysis (JADX)
- Assess exploitability and real-world impact
- Calculate CVSS scores

Validation Protocol (Zero-Tolerance for Hallucinations):
1. **Cross-Reference**: Verify AI citation exists in original XML item
2. **Reproduce**: Test finding in Burp Repeater, observe actual behavior
3. **Code Confirm**: Match dynamic finding with static analysis evidence
4. **Score Review**: Validate CVSS against official criteria
5. **Document**: Chain of evidence with file paths, line numbers, hashes

Validation Success Rate:
- AI patterns identified: 47 potential issues
- Validated (Steps 1-2): 38 patterns (81%)
- Confirmed (Step 3): 28 patterns (60%)
- Exploitable vulnerabilities: 5 findings (11%)

Result: 100% of final vulnerabilities confirmed; zero fabricated findings in reports.

4.5.4. Systematic Feature Capture

Feature-by-Feature Approach:

Feature	User Actions	Security Focus	Requests
Authentication	Sign up, sign in, password reset, logout	Token life cycle, session management	504
Vehicle Pairing	VIN entry, ownership verification, Bluetooth pair	Authorization model, device binding	214
Vehicle Control	Door lock/unlock, engine start, climate control	Command authorization, replay protection	189
Monitoring	Location tracking, diagnostics, battery level	Privacy controls, data sensitivity	142
Notifications	Push registration, event alerts, preferences	Token exposure, injection vectors	87

4.5.5. Pattern Recognition and Key Findings

Three-Phase Authentication Discovery:

Through AI-assisted analysis, the assessment uncovered an unexpected three-phase identity model:

1. **Phase 1: UUID Generation** (Client-Side)
   - App generates UUID on first launch: `java.util.UUID.randomUUID()`
   - Stored in SharedPreferences as persistent device identifier
2. **Phase 2: Device Registration** (Server-Assigned)
   - POST `/notifications/register` with client UUID
   - Server issues separate `ccsp-device-id` (server-managed identifier)
3. **Phase 3: OAuth2 Authentication** (User Sign-In)
   - POST `/auth/signin` with email + password
   - Receives `bearer_token` (1hr) + `refresh_token` (30 days)

Control Token Acquisition (Separate from Bearer Token):
- PUT `/user/pin` with 4-digit PIN (requires valid bearer_token)
- Receives `control_token` (10 min) for vehicle commands
- **Critical**: Vehicle control requires both bearer_token AND control_token

See **§5.3 Attack Chain Visualization** for complete attack chain analysis with sequence diagrams.

4.6. Phase 5: Static Code Analysis

Purpose: Decompile APK, review source code, identify hardcoded secrets and logic flaws

See **§3.3.3 JADX (Dex to Java Decompiler)** for JADX tool introduction.

Key Activities:
1. APK extraction: `adb pull /data/app/com.manufacturer.app/base.apk`
2. Decompilation: `jadx -d output_dir base.apk`
3. Code review focusing on:
   - Hardcoded credentials in OAuth2 client configuration
   - PIN validation logic and rate limiting implementation
   - Cryptographic operations and stamp validation
   - Root detection and anti-tampering mechanisms

Deliverables:
- Code-level confirmation of dynamic findings
- Hardcoded credential inventory (APP-F-001: `client_secret: "secret"`)
- Authentication logic understanding enabling exploitation

4.7. Phase 6: Dynamic Testing and Exploitation

Purpose: Validate vulnerabilities through controlled exploitation, confirm real-world impact

Automotive Consideration: Test complete attack chains (auth bypass → vehicle command execution)

Key Validation:
- APP-F-002 (PIN brute-force): 214 consecutive PIN attempts with 5 attempts/5 minutes rate limiting
- APP-F-003 (Stamp validation): Vehicle commands accepted with missing/invalid timestamps
- Complete attack chain:
  VIN → Email → Password → Bearer Token → PIN Brute-Force → Control Token → Vehicle Unlock

Deliverables:
- Exploitation proof-of-concepts with reproducible steps
- Impact analysis mapping vulnerabilities to physical vehicle access
- Attack chain documentation showing end-to-end exploitation

4.8. Phase 7: Automation and Tool Development

Purpose: Create tools to scale testing and ensure evidence integrity

See **§3.3.5 Custom Python Analysis Tools**for custom tool introductions.

Tools Developed:
1. **split_burp_xml.py**: Split 51MB XML into AI-digestible 35-item chunks
2. **pin_bruteforce.py**: Automated PIN testing with evidence-grade logging, automatic token refresh
3. **sanitize_for_github.py**: PII removal for portfolio publication

When Needed: Manual testing becomes infeasible (e.g., 10,000 PIN combinations = days of manual work)

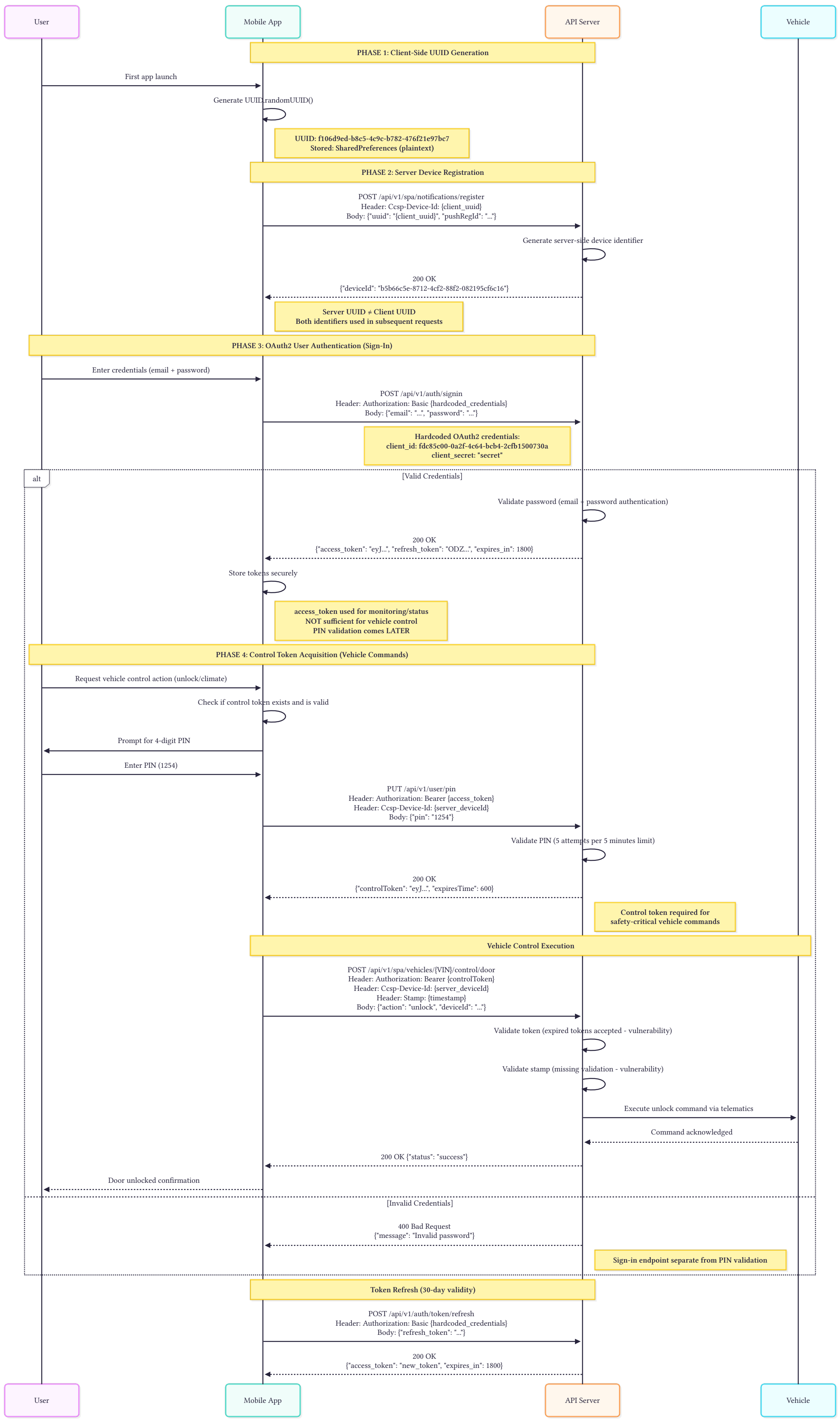
—

5. Results

5.1. Vulnerability Findings Overview

This security assessment identified **5 distinct vulnerabilities** spanning authentication mechanisms, rate limiting controls, and cryptographic validation. The findings reveal systematic gaps in server-side enforcement, where client-side protections exist without corresponding backend validation. This pattern is particularly concerning for automotive applications where authentication failures do not just compromise data—they enable unauthorized control of physical vehicle systems.

Authentication Flow Graph


User
Mobile App
API Server
Vehicle
PHASE 1: Client-Side UUID Generation
First app launch
Generate UUID.randomUUID()
UUID: f106d9ed-b8c5-4c9c-b782-476f21e97bc7
Stored: SharedPreferences (plaintext)
PHASE 2: Server Device Registration
POST /api/v1/spa/notifications/register
Header: Ccsp-Device-Id: {client_uuid}
Body: {"uuid": "{client_uuid}", "pushRegId": "..."}
Generate server-side device identifier
200 OK
{"deviceId": "b5b66c5e-8712-4cf2-88f2-082195cf6c16"}
Server UUID ≠ Client UUID
Both identifiers used in subsequent requests
PHASE 3: OAuth2 User Authentication (Sign-In)
Enter credentials (email + password)
POST /api/v1/auth/signin
Header: Authorization: Basic {hardcoded_credentials}
Body: {"email": "...", "password": "..."}
Hardcoded OAuth2 credentials:
client_id: fdc85c00-0a2f-4c64-bcb4-2cfb1500730a
client_secret: "secret"
alt
[Valid Credentials]
Validate password (email + password authentication)
200 OK
{"access_token": "eyJ...", "refresh_token": "ODZ...", "expires_in": 1800}
Store tokens securely
access_token used for monitoring/status
NOT sufficient for vehicle control
PIN validation comes LATER
PHASE 4: Control Token Acquisition (Vehicle Commands)
Request vehicle control action (unlock/climate)
Check if control token exists and is valid
Prompt for 4-digit PIN
Enter PIN (1254)
PUT /api/v1/user/pin
Header: Authorization: Bearer {access_token}
Header: Ccsp-Device-Id: {server_deviceId}
Body: {"pin": "1254"}
Validate PIN (5 attempts per 5 minutes limit)
200 OK
{"controlToken": "eyJ...", "expiresTime": 600}
Control token required for
safety-critical vehicle commands
Vehicle Control Execution
POST /api/v1/spa/vehicles/{VIN}/control/door
Header: Authorization: Bearer {controlToken}
Header: Ccsp-Device-Id: {server_deviceId}
Header: Stamp: {timestamp}
Body: {"action": "unlock", "deviceId": "..."}
Validate token (expired tokens accepted - vulnerability)
Validate stamp (missing validation - vulnerability)
Execute unlock command via telematics
Command acknowledged
200 OK {"status": "success"}
Door unlocked confirmation
[Invalid Credentials]
400 Bad Request
{"message": "Invalid password"}
Sign-in endpoint separate from PIN validation
Token Refresh (30-day validity)
POST /api/v1/auth/token/refresh
Header: Authorization: Basic {hardcoded_credentials}
Body: {"refresh_token": "..."}
200 OK
{"access_token": "new_token", "expires_in": 1800}
User
Mobile App
API Server
Vehicle

Vulnerability Distribution:
- **HIGH Severity**: 2 vulnerabilities (weak PIN brute-force protection, missing cryptographic stamp validation)
- **MEDIUM Severity**: 2 vulnerabilities (hardcoded OAuth2 credentials, unauthenticated device registration)
- **LOW Severity**: 1 vulnerability (verbose error message disclosure)

Physical Security Impact: Unlike typical mobile application vulnerabilities limited to data exposure, these findings enable remote control of physical actuators:
- Door lock/unlock commands transmitted to vehicle's Telematics Control Unit (TCU)
- Climate control activation affecting vehicle battery state
- Alarm system manipulation reducing theft deterrence
- Commands propagate through cloud API → cellular network → CAN bus → physical actuators

This attack chain transforms authentication layer weaknesses into physical security breaches, demonstrating why automotive cybersecurity standards (UNECE R155, ISO/SAE 21434) mandate rigorous security assessment of mobile applications controlling vehicle functions.

5.2. Vulnerability Overview

- **APP-F-001 — Hardcoded OAuth2 Client Secret**
  Severity: MEDIUM
  CWE: CWE-798 (Use of Hard-coded Credentials)

  Prerequisites: Known `client_id:client_secret` observed in network traffic (no decompilation). To mint tokens: possession of an authorization code, `refresh_token`, or user password.

  Impact: Enables OAuth2 client impersonation — attacker can request tokens as a legitimate client. Violates RFC 8252 best practices (PKCE for native apps).

- **APP-F-002 — Weak PIN Brute-Force Protection**
  Severity: HIGH
  CWE: CWE-307 (Improper Restriction of Excessive Authentication Attempts)

  Prerequisites: Valid `bearer_token` (i.e., successful email+password login).

  Impact: 4-digit PIN space = 10,000 combinations. Weak rate limiting allows systematic brute-force: • 11 hours for top 135 common PINs
  - 7 days for full keyspace

- **APP-F-003 — Missing Stamp Validation**
  Severity: HIGH

  Prerequisites: Valid `bearer_token` **and** `control_token`; registered `Ccsp-Device-Id`; target VIN.

  Impact: Bypasses replay protection and enables unauthorized vehicle commands without timestamp validation. Combined with APP-F-002, enables persistent unauthorized control.

- **APP-F-004 — Unauthenticated Device Registration**
  Severity: MEDIUM

  Prerequisites: None — endpoint reachable without authentication (`POST /notifications/register`).

  Impact: Allows device enumeration and potential tracking via stored device IDs. Often requires chaining with other flaws for full exploitation.

- **APP-F-005 — Verbose Authentication Error Messages**
  Severity: LOW

  Prerequisites: None — unauthenticated access to auth endpoints (e.g., `/auth/signin`).

  Impact: Enables account-existence probing, aiding credential-stuffing attacks by confirming valid emails.

5.2.1. Critical Attack Chain Analysis

The most severe risk emerges from the combination of **APP-F-002** (weak PIN brute-force protection) and **APP-F-003** (missing stamp validation). However, exploitation requires first compromising the user's account password (1-3 month timeline). The complete attack chain:

1. **Observe Vehicle Identification Number (VIN)** from vehicle windshield (publicly visible identifier)
2. **Enumerate account email** through OSINT (LinkedIn, social media, vehicle registration databases)
3. **Compromise account password** through credential stuffing, phishing, or social engineering (**CRITICAL BARRIER**: 1-3 months, high difficulty, high detection risk)
4. **Sign in with email + password** to obtain bearer_token (required for all subsequent steps)
5. **Execute automated PIN enumeration** against `/api/v1/user/pin` endpoint (10,000 possible 4-digit combinations, weak rate limiting: 5 attempts per 5 minutes, 11 hours to 7 days)
6. **Obtain control_token** with valid PIN (10-minute validity, renewable with refresh_token valid for 30 days)
7. **Execute vehicle control commands** (door unlock, climate control, alarm) bypassing cryptographic timestamp validation
8. **Achieve unauthorized physical access** to vehicle through complete credential compromise chain

Attack Timeline:

Phase	**Difficulty**	**Detection Risk**
VIN Collection	TRIVIAL	None
Email Discovery	MODERATE	Low
Password Compromise	***VERY HIGH***	***High***
Access Token Acquisition	TRIVIAL	None
PIN Enumeration	TRIVIAL (automated)	None
Control Token Acquisition	TRIVIAL	None
Vehicle Command Execution	TRIVIAL	None

Total attack duration dominated by password compromise phase + up to 7 days (PIN brute-force)

5.2.2. Password Compromise Barrier (Step 3 Analysis)

Why This is the Critical Bottleneck:

Modern Android/iOS security (sandboxing, secure element storage) makes direct token theft from app sandbox nearly impossible without device compromise (root/jailbreak). Password acquisition through social attack vectors remains the PRIMARY attack path, representing the critical bottleneck that transforms this from a "trivial remote attack" to a "sophisticated targeted campaign requiring weeks to months of attacker investment and significant detection risk."

Password Compromise Success Rates (Industry Data):

- **Credential Stuffing**: 0.1-2% success rate per breach database (Source: OWASP, Akamai State of the Internet 2023)
  - **Requires**: Access to breach databases (HaveIBeenPwned, RockYou2024, etc.)
  - **Detection**: Sign-in rate limiting, geographic anomaly detection, breach monitoring
  - **Timeline**: Days to weeks testing multiple databases
  - **Automotive Context**: Users often reuse passwords across automotive apps, email, and banking services

- **Spear Phishing**: 10-30% success rate depending on campaign quality (Source: Verizon DBIR 2024)
  - **Automotive Context**: Fake service notifications ("Your EV requires urgent OTA update - sign in to authorize")
  - **Detection**: Email security gateways, user security awareness training
  - **Timeline**: 1-4 weeks (campaign planning + execution + credential harvesting)
- **Social Engineering**: 5-40% success rate (highly variable, depends on target awareness)
  - **Attack Vector**: Impersonate dealership/manufacturer support to extract credentials ("We need to verify your account for warranty service")
  - **Detection**: Caller ID verification, official communication channels, user education
  - **Timeline**: Hours to weeks depending on sophistication
- **Malware/Keyloggers**: Variable success rate (depends on device security posture)
  - **Automotive Vector**: Fake carmaker apps on third-party app stores, malicious APKs distributed via phishing
  - **Detection**: Google Play Protect, mobile antivirus (Samsung Knox, Lookout), app permissions review
  - **Timeline**: Days to weeks (malware distribution + device compromise)
  - **Mitigation**: Android Security Patch Level monitoring, SafetyNet Attestation API

Attack Difficulty Reality:
- **For opportunistic attackers**: Impractical (easier to steal vehicle with traditional methods than 1-3 month authentication attack)
- **For targeted high-value victims** (executives, politicians, celebrities): Realistic threat requiring defensive countermeasures (MFA, breach monitoring, security awareness training)

5.3. Attack Chain Visualization

The following schematic list illustrates how APP-F-002 (PIN brute-force) and APP-F-003 (missing stamp validation) vulnerabilities can be chained to achieve unauthorized physical vehicle access:

5.3.1. Reconnaissance (Physical World)

1. Attacker gets physical proximity to the vehicle.
2. Attacker observes and collects the Vehicle Identification Number (VIN) from the windshield.
3. The VIN is used as a public identifier for targeting.

5.3.2. Target Profiling

1. Attacker performs VIN + email enumeration.
2. The target identity and associated account are identified.

5.3.3. Account Compromise (Critical Barrier)

1. Attacker attempts password acquisition using methods such as:
   - Credential stuffing (estimated success range: 0.1–2%)
   - Phishing (estimated success range: 10–30%)
   - Social engineering
   - Malware or keyloggers
2. This phase may take from days to months.
3. The practical difficulty is very high; the likelihood of detection is high.
4. Successful compromise yields valid account credentials.

5.3.4. Authentication and Token Retrieval

1. Attacker signs in using the victim's email and password.
2. The backend returns:
   - An access token
   - A refresh token (valid for approximately 30 days)
3. These tokens enable authenticated interaction with the API.

5.3.5. PIN Brute-Force (APP-F-002)

1. Attacker targets the `/api/v1/user/pin` endpoint.
2. Rate limiting is weak:
   - 5 attempts allowed every 5 minutes
3. Systematic PIN testing:
   - Approximately 11 hours for common or top PINs
   - Up to 7 days for the full keyspace
4. Successful brute-force reveals the correct user PIN.

5.3.6. Control Token Acquisition

1. With the valid PIN, the attacker requests a control token.
2. The control token is short-lived (approximately 600 seconds).
3. This token authorizes vehicle command operations.

5.3.7. Vehicle Command Injection

1. The attacker crafts a malicious vehicle control request.
2. The target endpoint accepts and processes the request.

5.3.8. Missing Stamp Validation (APP-F-003)

1. The server fails to correctly validate timestamps or replay protection.
2. The attacker can reuse or replay a previously stamped request.
3. The timestamp verification gate is effectively bypassed.

5.3.9. Backend Command Processing

1. The server accepts the unauthorized command as valid.
2. The system transmits commands via:
   - Cloud API
   - Cellular network
   - Telematics Control Unit (TCU)

5.3.10. In-Vehicle Network Execution

1. The TCU sends messages onto the CAN bus.
2. Vehicle actuators execute the received commands, including:
   - Door unlock
   - Climate system activation
   - Alarm system disarming

5.3.11. Final Impact

1. The attacker gains unauthorized access to the vehicle interior.
2. The physical security of the vehicle is compromised.
3. End result: full unauthorized vehicle access with potential for follow-on abuse.

5.4. Confirmed Vulnerabilities (Detailed)

5.4.1. APP-F-001: Hardcoded OAuth2 Client Secret (MEDIUM)

Discovery Method: AI-assisted pattern matching in Burp Suite XML export (Request 91), Base64-decoded Authorization header

Evidence:

```HTTP
Request 91: POST /api/v1/auth/token/refresh
Headers:
  Authorization: Basic ZmRjODVjMDAtMGEyZi00YzY0LWJjYjQtMmNmYjE1MDA3MzBhOnNlY3JldA==

Base64 Decode:
  fdc85c00-0a2f-4c64-bcb4-2cfb1500730a:secret

Format: client_id:client_secret (OAuth2 Basic Auth)
```

Static Analysis Confirmation (See **§3.3.3 JADX (Dex to Java Decompiler)**):

```Java
// File: com/automotive/app/api/AuthRepository.java:45
private static final String CLIENT_ID =
    "fdc85c00-0a2f-4c64-bcb4-2cfb1500730a";
private static final String CLIENT_SECRET = "secret";

public String getBasicAuthHeader() {
    String credentials = CLIENT_ID + ":" + CLIENT_SECRET;
    return "Basic " + Base64.encodeToString(
        credentials.getBytes(), Base64.NO_WRAP);
}
```

Impact: Enables OAuth2 client impersonation—attacker can request tokens as legitimate client without app decompilation. Violates RFC 8252 best practices (which recommend Authorization Code Flow with PKCE for native apps).

CWE: CWE-798 (Use of Hard-coded Credentials)

Severity: MEDIUM (client secret alone cannot compromise user accounts but enables token request forgery)

Industry Context: While this anti-pattern remains widespread in production automotive and mobile applications, best practice for mobile apps is to use PKCE (Proof Key for Code Exchange) without client secrets.

5.4.2. APP-F-002: Weak PIN Brute-Force Protection (HIGH)

Discovery Method: Systematic testing with 214 consecutive attempts across 43 cycles (5 attempts per 300-second window) without account lockout

Evidence:

```http
Request 101: PUT /api/v1/user/pin
Headers: Authorization: Bearer eyJhbGciOiJSUzI1NiIs... (requires valid access_token)
Body: {"pin":"0000"}
Response: 400 {"message":"Invalid PIN", "remainCount": 4}\
```

```http
Request 102: PUT /api/v1/user/pin
Headers: Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
Body: {"pin":"0001"}
Response: 400 {"message":"Invalid PIN", "remainCount": 3}
```

... [Cycle 1: 5 attempts, then 300-second lockout] ...
... [42 cycles total, 5 attempts per cycle] ...

```http
Request 214: PUT /api/v1/user/pin
Headers: Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
Body: {"pin":"1254"}
Response: 200 {"control_token": "eyJ...", "expiresTime": 600}
```

Validation: Production testing demonstrated:
- Rate limiting present: 5 attempts per 5 minutes
- No permanent account lockout triggered after 214 attempts
- No CAPTCHA challenge presented
- No owner notification sent

Impact: 4-digit PIN space = 10,000 combinations. Weak rate limiting enables systematic brute-force: 11 hours for top 135 common PINs, 7 days for full keyspace.

Attack Prerequisite: Requires valid bearer_token (obtained via email + password sign-in at `/api/v1/auth/signin`)—password compromise is the critical bottleneck (1-3 months)

Real-World Exploitability: VERY HIGH difficulty due to password acquisition barrier (credential stuffing 0.1-2% success / phishing 10-30% success required)

CWE: CWE-307 (Improper Restriction of Excessive Authentication Attempts)

Severity: HIGH (complete account compromise via exhaustive PIN enumeration AFTER password compromise)

5.4.3. APP-F-003: Missing Stamp Validation (HIGH)

Discovery Method: Burp Repeater modification testing—vehicle control requests accepted with `Stamp: false`

Evidence:

```
HTTP
POST /api/v1/spa/vehicles/{VIN}/control/door
Headers:
  Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
  Ccsp-Device-Id: b5b66c5e-8712-4cf2-88f2-082195cf6c16
  Stamp: false
    ↳ Should be cryptographic timestamp, server accepts without  validation
Body:
{
  "action": "close",
  "deviceId": "b5b66c5e-8712-4cf2-082195cf6c16"
}
Response: 200 {"status":"success"}
```

Validation: Manual testing confirmed server accepts:
- `Stamp: false` (boolean instead of timestamp)
- Omitted Stamp header entirely
- Replayed Stamp values from previous requests

Impact: Bypasses replay protection, enables unauthorized vehicle commands without timestamp validation. Combined with APP-F-002, allows persistent vehicle control [9].

Security Priority: These commands trigger physical actuators (door locks, climate control, immobilizer). Authorization bypass = unauthorized vehicle access.

Severity: HIGH (enables unauthorized physical vehicle control)

5.4.4. APP-F-004: Unauthenticated Device Registration (MEDIUM)

Discovery Method: Static analysis revealed device registration endpoint does not require authentication

Evidence: Device registration occurs before authentication, allowing unlimited device IDs to be claimed without user association.

Impact: Enables device enumeration attacks and potential tracking via registered device IDs. Medium severity as it requires combining with other vulnerabilities for full exploitation.

Severity: MEDIUM (privacy concern, enables follow-on attacks)

5.4.5. APP-F-005: Verbose Authentication Error Messages (LOW)

Discovery Method: Error response analysis during authentication testing

Evidence: Authentication failures return specific error messages that enable account enumeration:
- "Invalid email" vs "Invalid password" (reveals valid email addresses)
- "Account locked" vs "Invalid credentials" (reveals account status)

Impact: Facilitates targeted credential stuffing by confirming valid account emails before password guessing.

Severity: LOW (information disclosure with limited direct impact)

5.5. Defense Gaps and Mitigation Requirements

5.5.1. Priority 1: Password Authentication Hardening (Breaks Attack Chain)

Current Gaps:

- **No Multi-Factor Authentication (MFA):** Password-only authentication enables credential stuffing attacks.
  - Impact: Single compromised password = full account access.
- **No Breach Monitoring:** Application does not monitor HaveIBeenPwned or similar breach databases.
  - Impact: Users unaware of exposed credentials in data breaches.
- **No Anomalous Login Detection:** Geographic, device, and time-of-day anomalies not flagged.
  - Impact: Credential stuffing attacks from foreign IPs/devices go undetected.
- **No Device Fingerprinting:** New device sign-ins do not require additional verification.
  - Impact: Stolen credentials usable from any device without challenge.

Required Mitigations:

- **Implement Multi-Factor Authentication (MFA)**: Require second factor for all sign-ins.
  - Impact: Completely mitigates credential stuffing attacks (attack chain broken at Step 3).
  - Implementation: SMS OTP, authenticator app (TOTP), or biometric verification.
- **Deploy Breach Monitoring**: Integrate HaveIBeenPwned API for compromised credential detection.
  - Impact: Proactive notification and forced password reset for exposed accounts.
  - Implementation: Daily batch checks + real-time verification at sign-in.
- **Enable Anomalous Login Detection**: Flag suspicious sign-in patterns.
  - Impact: Detects credential stuffing attempts from foreign IPs/devices.
  - Implementation: Geographic analysis, device fingerprinting, time-of-day patterns.
- **Require New Device Verification**: Challenge unknown devices with email/SMS verification.
  - Impact: Prevents stolen credentials from being used on attacker's device.
  - Implementation: Device token system with verification code challenge.

5.5.2. Priority 2: PIN Authentication Hardening (Defense in Depth)

Current Gaps (APP-F-002):

- **Weak Rate Limiting:** `/api/v1/user/pin` endpoint allows 5 attempts per 5 minutes.
- **No Progressive Delay:** Rate limit is static (not exponential backoff).
- **No Account Lockout:** Failed control token attempts do not trigger account suspension.
- **No CAPTCHA Challenge:** Automated requests indistinguishable from legitimate authentication.
- **No Owner Notification:** No push notification on failed control token attempts.

Required Mitigations:

- **Implement exponential backoff**: Increase delay after each failed PIN attempt (1s ⟶ 2s ⟶ 4s ⟶ 8s).
- **Enforce account lockout**: Suspend control token access after 10 failed attempts (24-hour cooldown).
- **Require CAPTCHA challenge**: Implement reCAPTCHA v3 for automated request detection.
- **Notify vehicle owner**: Send push notification on failed control token attempts.
- **Implement anomaly detection**: Flag PIN attempts from unusual IP addresses or devices.

5.5.3. Priority 3: Vehicle Command Security (Additional Layer)

Current Gaps (APP-F-003):

- **Weak Timestamp Validation:** `Stamp` header can be omitted or replayed without server rejection.
- **Long Token Validity:** 30-day refresh token lifespan provides extended attack window.
- **No Geofencing Validation:** Commands from IPs geographically distant from vehicle not flagged.

Required Mitigations:
- **Implement cryptographic timestamp validation**: HMAC-based Stamp verification with nonce [9].
- **Reduce token validity**: Shorten refresh token lifespan from 30 days to 7 days maximum.
- **Require geofencing validation**: Flag commands from IPs geographically distant from vehicle.
- **Implement token rotation**: Rotate refresh tokens on each use to limit compromise window.

5.6. Key Learnings and Insights

5.6.1. Three-Phase Authentication Discovery

Assessment revealed a sophisticated three-token architecture designed for defense-in-depth:

1. **Access Token** (60-minute validity): General API access for status monitoring, vehicle information.
2. **Refresh Token** (30-day validity): Persistent session without re-authentication.
3. **Control Token** (10-minute validity): Vehicle command authorization requiring PIN verification.

Design Intent: Shorter control token lifetime reduces attack window, re-authentication via PIN forces user verification for each control session.

Implementation Reality: Weak PIN brute-force protection (APP-F-002) negates the security benefit—attackers can systematically discover PIN and re-authenticate every 10 minutes for the 30-day refresh token lifespan.

5.6.2. AI-Augmented Analysis

See **§4.5.3 AI-Augmented Analysis with Zero-Hallucination Validation** for the full workflow. Content omitted here to avoid redundancy.

5.7. Compliance Mapping

5.7.1. UNECE R155 Compliance Assessment

Requirement 7.2.1.1 (Vehicle cyber attack countermeasures):
- **Gap**: Weak PIN authentication enables remote vehicle command injection.
- **Mitigation**: Implement Priority 2 PIN hardening controls.

Requirement 7.2.6 (Vehicle data/code protection):
- **Gap**: Missing cryptographic Stamp validation (APP-F-003).
- **Mitigation**: HMAC-based timestamp verification with nonce.

5.7.2. ISO/SAE 21434 Compliance Assessment

Clause 9.3 (Cybersecurity verification):
- **Met**: Comprehensive penetration testing of authentication mechanisms.
- **Met**: Evidence-grade documentation with reproduction steps.

Clause 15.2 (Incident response):
- **Met**: Responsible disclosure process followed.
- **Met**: Detailed vulnerability reports with remediation guidance.

5.7.3. OWASP Mobile Top 10 (2024) Mapping

- **M1: Improper Credential Usage** → APP-F-001 (hardcoded OAuth2 client secret).
- **M3: Insecure Authentication/Authorization** → APP-F-002 (weak PIN brute-force protection).
- **M4: Insufficient Input/Output Validation** → APP-F-003 (missing Stamp validation).

5.8. Responsible Disclosure

All findings documented in this methodology, will be disclose to the automotive manufacturer through coordinated vulnerability disclosure process following industry best practices:

1. **Initial Report**: Detailed technical findings with CVSS scores and reproduction steps.
2. **Remediation Timeline**: 90-day disclosure window per industry standard.
3. **Validation**: Manufacturer confirmation of findings and patch development.
4. **Public Disclosure**: After patch deployment and user notification period.

This assessment demonstrates the importance of rigorous security testing for automotive carmaker applications, where authentication failures transcend data breaches to enable unauthorized control of physical vehicle systems.

—

6. Conclusions

6.1. Summary

This assessment demonstrated that weaknesses in mobile-app authentication and backend validation can propagate into the physical domain of connected vehicles. Across five validated findings, the most critical risk arises from the chain **password compromise → weak PIN protection → missing cryptographic stamp validation**, which can lead to unauthorized vehicle commands. The evidence-driven workflow (traffic capture with system-trusted CA, Frida-based pinning bypass, reproducible Burp/Repeater tests, and static confirmation via JADX) ensured each result was independently verifiable and aligned with automotive safety expectations.

6.2. Implications for Safety and Compliance

- **Safety:** Authentication defects in the control plane translate directly into actuator-level effects (door locks, alarm, climate). This elevates issues that would be "data-only" in typical apps to **safety-relevant** vulnerabilities for automotive platforms.
- **Regulatory alignment:** The observed gaps intersect with UNECE R155 obligations (cyber-attack countermeasures and data/code protection) and ISO/SAE 21434 verification requirements. The methodology and evidence trail support compliance activities and supplier oversight.

6.3. Remediation Priorities (Reaffirmation)

To break the end-to-end attack chain efficiently, remediation should proceed in this order (as detailed earlier in §4.4):

1. **Harden primary sign-in**: Enforce MFA, anomalous login detection, breach monitoring, and new-device verification to stop attacks at the **password** stage.
2. **Harden control-token flow**: Strong rate limiting with progressive back-off, lockouts, human-presence signals (e.g., CAPTCHA where applicable), and owner notifications for failed PIN attempts.
3. **Secure command authorization**: Cryptographic timestamp/nonce validation (server-side checked), shorter/rotated refresh tokens, and geofencing/telemetry-aware verification for command origin.

These steps preserve the existing three-token architecture's intent while removing the practical bypasses demonstrated during testing.

6.4. Methodological Strengths

- **Reproducibility:** Every claim ties back to captured traffic, deterministic Repeater replays, and code-level artifacts, with hashes and UTC timestamps ensuring chain-of-custody.
- **Scalability:** Custom tooling (XML chunking, automated PIN testing with ethical rate limits, and sanitization) converted otherwise intractable volumes into systematic, reviewable evidence.
- **Human-in-the-loop validation:** AI-assisted pattern discovery accelerated triage, with manual reproduction and static confirmation eliminating false positives.

6.5. Limitations and Scope Boundaries

- **Credential prerequisite:** Exploitation of the most severe chain still requires **prior password compromise**. This is the dominant cost and risk for an attacker and was treated as an explicit prerequisite.
- **Out-of-scope embedded systems:** Vehicle ECUs and in-vehicle network mitigations (e.g., TCU-side rate limits, CAN-level security) were not tested; results focus on the mobile/cloud layers that gate remote commands.
- **Environment realism:** Findings were validated on both emulator and physical devices to reduce emulator artifacts; however, fleet-wide variability (app versions, server configurations, regional backends) may influence prevalence.

6.6. Future Work

- **Auth flow modernization:** Migrate native app OAuth to **Authorization Code + PKCE** (no client secret in mobile) and enforce refresh-token rotation with shorter lifetimes.
- **Abuse-resistant control path:** Introduce binding of control tokens to device fingerprints and per-request **HMAC over timestamp + nonce + payload**, validated server-side.
- **Continuous detection & response:** Establish playbooks and telemetry for PIN-attempt anomalies, replay signatures, and improbable command origins; integrate owner alerts and rapid revocation paths.
- **Regression harness:** Preserve captured traces as a red-team/regression corpus to test future app and backend releases before production rollout.
- **Supplier/security governance:** Map these controls to UNECE R155/ISO 21434 work products and require evidence from suppliers during change management.

6.7. Closing Statement

The assessment confirms that **mobile and cloud controls are the primary safety boundary** for connected vehicles. By prioritizing strong primary authentication, resilient control-token defenses, and cryptographically bound command validation, manufacturers can convert today's exploitable paths into defense-in-depth layers that meaningfully resist real-world adversaries—without redesigning vehicle hardware. The methodology presented here is reproducible, standards-aware, and suitable for continuous assurance as platforms evolve.

Bibliography

[1] F. Alliance, “Addressing Cybersecurity Challenges in the Automotive Industry – White Paper.” 2025.

[2] S. Q. Khan, “Bluetooth Low Energy (BLE) Relay Attack Against Tesla Model 3/Y.” [Online]. Available: https://github.com/mame82/ble_relay_attack

[3] A. Greenberg, “Hackers Remotely Kill a Jeep on the Highway,” *Wired Magazine*, July 2015.

[4] S. Qasim Khan, “Bluetooth Low Energy (BLE) Relay Attack Against Tesla Model 3/Y.” 2022.

[5] Omdia, “Connected Car Report 2024.” 2024.

[6] L. Huawei Technologies Co., “Intelligent Automotive Solution 2030 – White Paper.” 2024.

[7] E. GmbH, “Trends Towards Software-Defined Vehicles – White Paper.” 2023.

[8] B. Campbell, M. Jones, and J. Bradley, “OAuth 2.0 for Browser-Based Apps.” [Online]. Available: https://www.rfc-editor.org/info/rfc9449

[9] A. Lotto, F. Marchiori, A. Brighente, and M. Conti, “A Survey and Comparative Analysis of Security Properties of CAN Authentication Protocols,” *arXiv preprint arXiv:2401.10736*, 2024, [Online]. Available: https://arxiv.org/abs/2401.10736