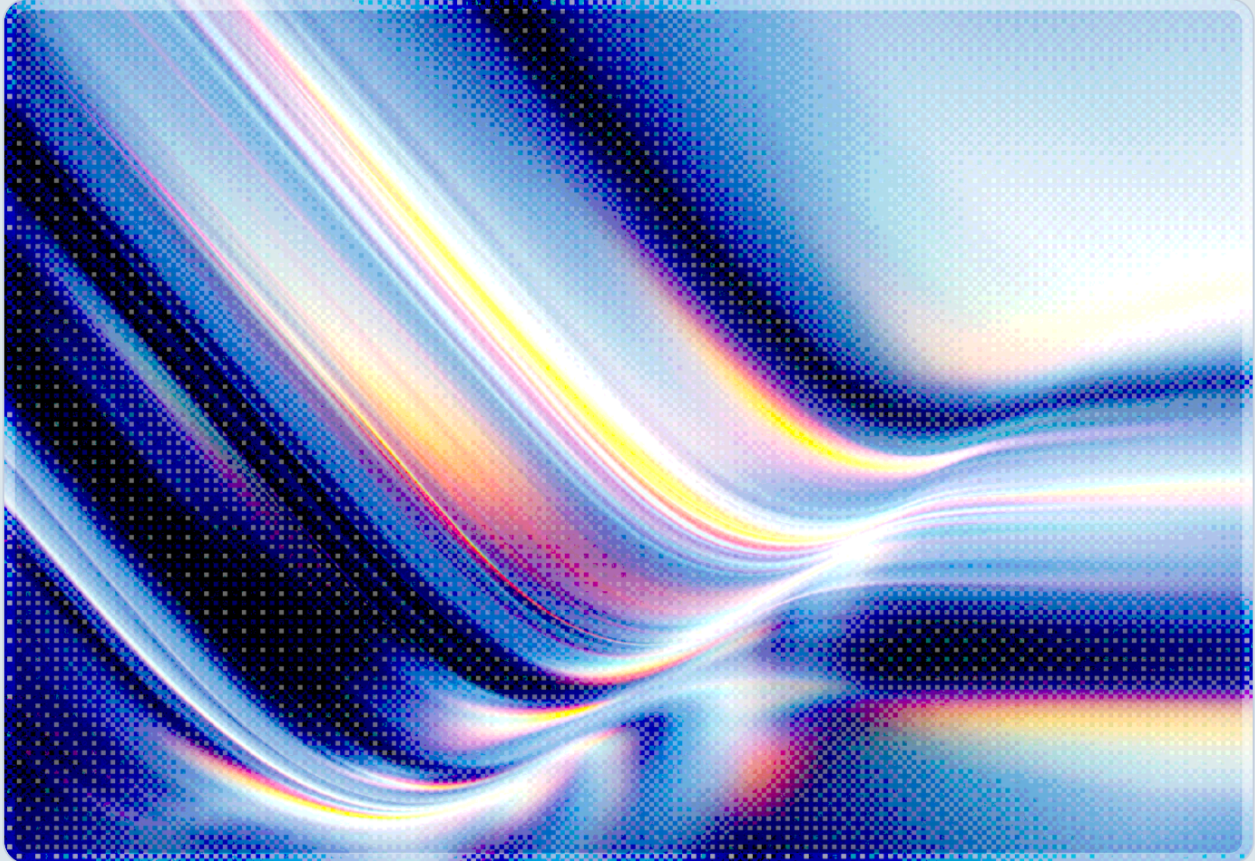


New Attack Surfaces in Third-Party Mobile AI Agents

Screen Perception and Channel Hijacking in ADB-Driven Agent Frameworks

2026-07-21

 AI-assisted Rapid Research



CSAI

cloud
CSA security
alliance®

© 2026 Cloud Security Alliance. Some rights reserved.

You may download, store, display, view, print, redistribute, and link to this document in its original, unmodified form, provided that attribution to the Cloud Security Alliance is maintained and all trademark and copyright notices remain intact.

This document may not be modified or altered. You may quote portions of the document as permitted by the Fair Use provisions of the United States Copyright Act, provided that attribution is given to the Cloud Security Alliance.

This document may be shared on professional and social media platforms in its original form with attribution.

This document was generated with AI assistance and has not undergone official CSA review and approval processes.

Key Takeaways

- Researchers evaluated five widely used open-source, third-party mobile AI agent frameworks – AppAgent, AppAgentX, Mobile-Agent-v3, Open-AutoGLM, and MobA – and found all five vulnerable to at least six of seven newly demonstrated attacks, none requiring root access or elevated Android permissions beyond what an ordinary third-party app can request [1].
- The attacks exploit a structural trust mismatch: these agents perceive a smartphone's screen only as a raw screenshot captured over ADB, so anything a malicious, co-installed app can render into that screenshot – including content invisible to a human user – is treated by the agent's vision-language model as ground truth [1].
- A demonstrated attack chain combines a near-invisible visual overlay with a screenshot-tampering race condition and shell metacharacter injection to achieve arbitrary command execution on the host PC or server running the agent, not merely on the phone itself [1].
- Several of the underlying weaknesses – unprotected broadcast receivers, world-readable temporary files, and shell commands built by string concatenation – are conventional Android and subprocess-handling defects that predate agentic AI, but the paper argues they become newly dangerous once an autonomous agent is given standing authority to act on what it perceives [1].
- The findings extend a growing body of research on environmental and prompt injection against mobile GUI agents, indicating this is a systemic class of risk across the ecosystem rather than an isolated flaw in any single framework [1][7][8].

Background

Mobile AI agents that operate a smartphone on a user's behalf – reading the screen, deciding what to tap, and issuing input – have moved rapidly from research prototypes to actively maintained open-source projects. Unlike agent features built directly into a device's operating system, the frameworks examined in this research are third-party: standalone applications, typically driven from an external host machine such as a PC or a server, that connect to an Android device over the Android Debug Bridge (ADB) to capture screenshots and inject taps, swipes, and text. AppAgent, AppAgentX, Mobile-Agent-v3, Open-AutoGLM, and MobA each follow this basic pattern, pairing a vision-language model with ADB-based perception and control to automate tasks ranging from app navigation to multi-step

workflows across mobile, web, and desktop surfaces [1][2][3][4][5][6]. Their appeal is straightforward: because they operate through the same screenshot-and-input interface a human uses, they can automate almost any app without requiring cooperation from that app's developer, a property that has driven adoption across research and practitioner communities as these projects have matured over the past two years.

That same property is also the source of the vulnerability class documented in the arXiv paper "(A)I Sees What You Don't: Exploiting New Attack Surfaces in Third-Party Mobile Agents," submitted July 1, 2026 and revised July 14, 2026 [1]. Because these agents never receive a structured, verified representation of the screen – no accessibility tree cross-check, no cryptographic binding between what is rendered and what is reported – their entire model of reality is a single screenshot and a channel of ADB commands, both of which sit within reach of any other app installed on the same device. The researchers frame this as a "fundamental trust mismatch in autonomous agent design": the agent's authority to act is calibrated to a screen it assumes is faithfully representing the state a human would see, but nothing in the architecture actually guarantees that assumption holds when an adversarial app is running concurrently in the background. This research arrives amid a broader wave of academic work probing mobile GUI agent robustness against environmental and content-based injection, underscoring that the weakness is conceptual rather than confined to any one codebase [7][8].

Security Analysis

The paper organizes its findings into two attack-surface categories. The first, screen perception attacks, exploits the gap between what a human eye can register and what a vision-language model will faithfully transcribe from a screenshot. The second, misused channel attacks, targets the ADB-based plumbing – screenshot capture, input injection, and command execution – that connects the agent's host process to the device. Across both categories, the researchers implemented and evaluated seven distinct attacks, summarized below.

Attack	Category	Mechanism
Subliminal visual injection	Screen perception	A near-transparent overlay (as low as 2% opacity in testing) renders instructions invisible to a human but legible to the agent's vision-language model
Invisible zone injection	Screen perception	Payloads are placed in display dead zones – rounded corners and camera cutouts – using standard Android display APIs,

Attack	Category	Mechanism
		areas a user rarely inspects but that remain in the captured screenshot
UI spoofing	Screen perception	A malicious app detects when a target app launches (via Accessibility Service) and overlays a convincing but fake interface to capture credentials the agent then transcribes or enters
Screenshot tampering	Misused channel	Exploits the timing window (measured at roughly 50–500ms, averaging about 210ms) between the device capturing a screenshot and the host pulling that file, substituting a modified image before the agent reads it
Broadcast-based input interception	Misused channel	Registers a receiver for unprotected system broadcasts used by some agents' input paths, allowing an installed app to eavesdrop on text the agent is entering, including credentials
Credential sniffing	Misused channel	An Accessibility Service listens for text-change events to capture plaintext input, including passwords, as the agent types
Host-side command injection	Misused channel	Shell metacharacters embedded in on-screen content are transcribed by the vision-language model into a command string that the host executes without sanitization

All five frameworks were vulnerable to the perception-based attacks (subliminal injection, invisible-zone injection, and UI spoofing) regardless of which vision-language model backed them, with success rates the paper reports as consistently at or near 20 out of 20 test trials across GPT-4o, Claude, Gemini, and smaller open models such as AutoGLM's own lightweight variant. Susceptibility to the channel-based attacks varied more by implementation choice. Open-AutoGLM's use of base64 encoding for its input path neutralized the shell-metacharacter injection attack, and MobA's use of an alternate screenshot-capture command narrowed, though did not eliminate, the tampering race window. Mobile-Agent-v3 and MobA, which rely on a hybrid input mechanism, were only exposed to broadcast interception for a subset of characters their primary input path could not handle directly. No framework was immune to the full attack set: Mobile-Agent-v3 was vulnerable to all seven attacks tested, while AppAgent, AppAgentX, Open-AutoGLM, and MobA were each vulnerable to six of seven. Notably, AppAgent and AppAgentX, which route all input through direct ADB shell commands rather than a hybrid approach, gained immunity

only to broadcast-based input interception as a side effect of that same design choice – their exclusive reliance on direct ADB text input leaves them just as exposed as the other frameworks to the remaining six attacks, including host-side command injection.

The paper's most consequential demonstration is a compound, end-to-end exploitation chain rather than any single attack in isolation. In a case study built around an agent operating WeChat, the researchers first used a subliminal visual overlay to inject a fabricated "network sync" message, which caused the agent to pause and effectively widened the tampering race window on its next screenshot capture. During that extended window, a substituted screenshot injected a payload containing shell metacharacters. The agent's vision-language model transcribed that payload into what it understood as ordinary on-screen text, and the host process – which the paper notes executed subprocess calls with `shell=True` rather than passing arguments as a structured list – concatenated the transcribed text directly into a command line, resulting in arbitrary code execution on the Windows or Linux machine running the agent. This detail matters beyond the specific frameworks tested: it demonstrates that a low-privilege Android app with no special permissions can, through an agent intermediary, escape the device entirely and compromise the host infrastructure the agent runs on, a considerably larger blast radius than a compromised phone alone.

The researchers conducted coordinated disclosure with the affected projects ahead of publication and, notably, withheld fully weaponized proof-of-concept code, releasing only sanitized demonstration artifacts and evaluation harnesses that require manual instrumentation to reproduce. The paper does not report CVE identifiers, which is consistent with the fact that several of the affected projects are research-oriented open-source repositories rather than commercially supported products with formal vulnerability-disclosure programs – a gap that itself complicates the remediation pathway for downstream adopters.

Recommendations

Immediate Actions

Organizations that have deployed, or are piloting, any of the five frameworks examined in this research – or architecturally similar ADB-driven mobile agents – should treat host-side command construction as the most urgent item to audit. Any code path that concatenates agent-transcribed screen text into a shell command string, particularly with `shell=True` in Python subprocess calls, should be rewritten to pass arguments as a structured list, eliminating the metacharacter injection vector regardless of what the agent perceives on screen. Teams should also confirm whether their agent implementation writes

screenshots to a world-readable or predictably named temporary file before consuming them; if so, that TOCTOU window should be closed by switching to in-memory streaming or an equivalent mechanism that avoids an intermediate file another app could race to modify.

Short-Term Mitigations

Beyond the host-side fix, defenders should apply the narrower countermeasures the paper proposes for each attack class: contrast-enhancement preprocessing before an image reaches the vision-language model to surface low-opacity subliminal text before the model does; software-level masking of display corner and cutout regions until device manufacturers provide a hardware-level solution; and signature-level permission protection on any broadcast receivers used for text input, replacing implicit broadcasts with explicit, package-scoped intents. Where an agent framework relies on Android's Accessibility Service for either UI-spoofing detection or credential handling, security teams should add foreground-activity monitoring that flags unexpected transitions and maintain an allowlist of expected package identifiers, since the paper's UI-spoofing attack depends on the malicious overlay appearing without triggering such a check.

Strategic Considerations

At a strategic level, this research argues for treating mobile GUI agents as a distinct category within agentic AI threat modeling rather than folding them into general LLM application security guidance. The paper's authors frame the core problem as an architectural one: these agents are granted standing authority to act on whatever a screenshot shows them, with no mechanism to verify that the screenshot reflects an unmanipulated device state. Closing that gap durably will likely require perception-aware security models purpose-built for multi-tenant device platforms – for instance, a verified or cryptographically attested screen-capture path that a malicious co-installed app cannot intercept or race – rather than a patchwork of the point mitigations above. Enterprises evaluating third-party mobile agent frameworks for internal automation should also weight vulnerability-disclosure maturity in vendor selection, since several of the frameworks examined here are community-maintained research projects without a formal security response process, meaning organizations that adopt them inherit responsibility for tracking and remediating this class of finding themselves.

CSA Resource Alignment

This research maps directly onto testing categories CSA has already published for agentic systems. CSA's *Agentic AI Red Teaming Guide* defines twelve threat categories for agentic AI, including Agent Critical System Interaction and Agent Authorization and Control Hijacking, both of which describe exactly the failure mode this paper demonstrates: an agent with standing authority over device and host-level actions that can be manipulated into executing unintended, attacker-controlled commands through its own perception pipeline [9]. The guide's structured, step-by-step testing methodology for command-injection and control-hijacking scenarios gives organizations operating mobile GUI agents a concrete framework for red-teaming exactly the attack chain this paper documents, including the compound perception-plus-channel exploitation pattern behind its host-RCE case study.

Because this is a device- and application-layer instance of a broader agentic threat class, CSA's *MAESTRO Agentic AI Threat Modeling* framework remains the appropriate reference for situating these findings within an organization's overall agent threat model, particularly for teams that have not yet formally modeled the trust boundary between an agent's perception layer and the environment it observes [10]. The *AI Controls Matrix (AICM v1.1)*, specifically its identity and access management and application-security domains, further supports the governance layer of remediation: enforcing least-privilege scoping for the host processes that execute agent-issued commands, and applying rigorous input-validation controls to any pipeline that transcribes untrusted, agent-perceived content into executable instructions [11]. Organizations building or deploying third-party mobile agent frameworks should treat these three CSA resources as a baseline reference set for both technical testing and governance controls addressing this vulnerability class.

References

- [1] arXiv. "[\(A\)I Sees What You Don't: Exploiting New Attack Surfaces in Third-Party Mobile Agents](#)." arXiv:2607.00333, submitted July 1, 2026, revised July 14, 2026.
- [2] TencentQQGYLab. "[AppAgent: Multimodal Agents as Smartphone Users](#)." GitHub, 2026.
- [3] Westlake-AGI-Lab. "[AppAgentX: Evolving GUI Agents as Proficient Smartphone Users](#)." GitHub, 2026.
- [4] X-PLUG. "[Mobile-Agent: The Powerful GUI Agent Family](#)." GitHub, 2026.
- [5] zai-org. "[Open-AutoGLM: An Open Phone Agent Model & Framework](#)." GitHub, 2026.
- [6] OpenDFM. "[MobA: Multifaceted Memory-Enhanced Adaptive Planning for Efficient Mobile Task Automation](#)." GitHub, 2026.
- [7] arXiv. "[MIRAGE: Context-Aware Prompt Injection against Mobile GUI Agents via User-Generated Content](#)." arXiv:2605.28116, 2026.
- [8] arXiv. "[GhostEI-Bench: Do Mobile Agents Show Resilience to Environmental Injection in Dynamic On-Device Environments?](#)" arXiv:2510.20333, 2026.
- [9] Cloud Security Alliance. "[Agentic AI Red Teaming Guide](#)." Cloud Security Alliance, 2025.
- [10] Cloud Security Alliance. "[Agentic AI Threat Modeling Framework: MAESTRO](#)." Cloud Security Alliance, February 6, 2025.
- [11] Cloud Security Alliance. "[AI Controls Matrix \(AICM v1.1\)](#)." Cloud Security Alliance, 2026.