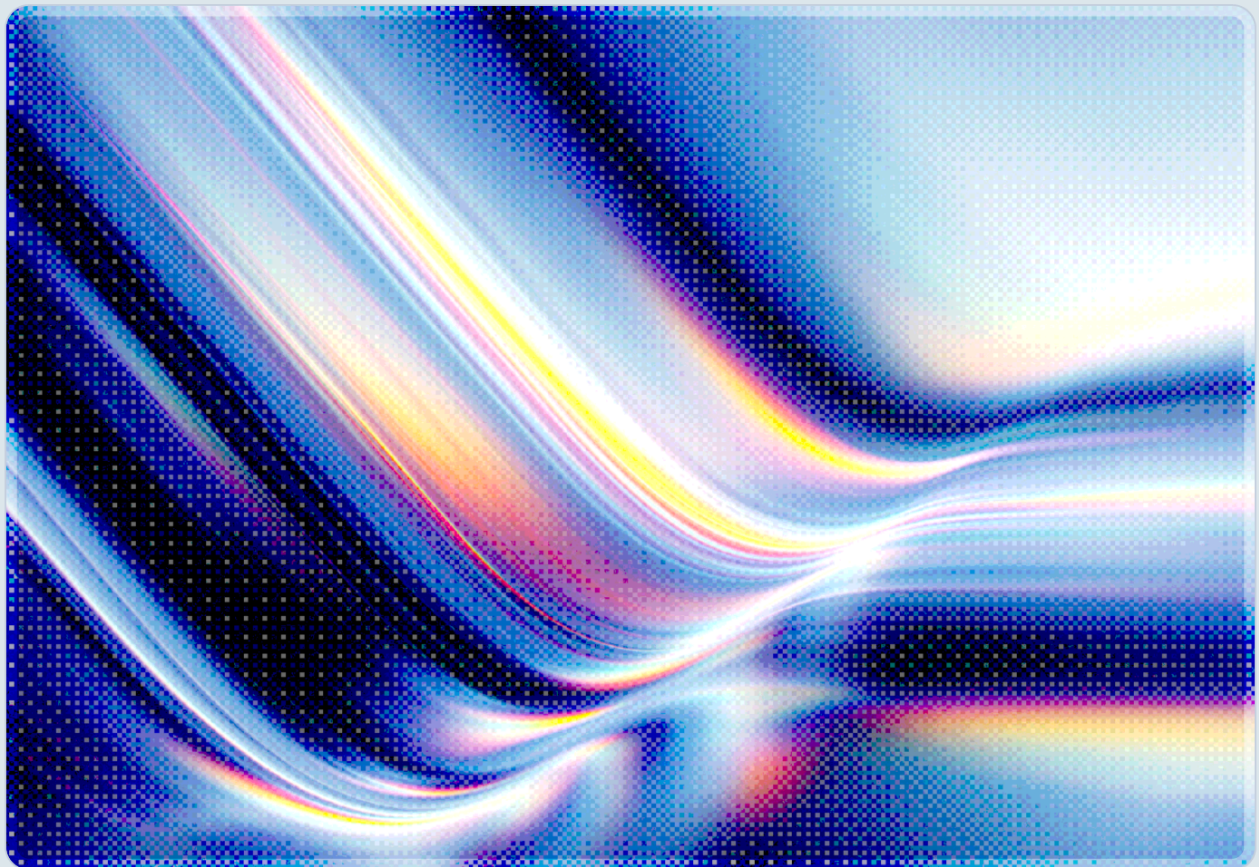


Invisible Screen Text Lets Malicious Apps Hijack Mobile AI Agents

2026-07-22

 AI-assisted Rapid Research



CSAI

cloud
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 demonstrated that an Android app needing only overlay and shared-storage access can paint on-screen text at 2-20% opacity that is invisible to a human user but reliably extracted by every vision-language model (VLM) tested – GPT-4o, Claude Opus 4.5, Gemini 3 Pro, GLM-4V, Qwen3-VL, and the lightweight AutoGLM-Phone all read the hidden text in 18 to 20 of 20 trials [1][2].
- Five widely used open-source mobile agent frameworks – AppAgent, AppAgentX, Mobile-Agent v3, Open-AutoGLM, and MobA – each fell to at least six of seven attack techniques the researchers built around this blind spot, including a race-condition attack that swaps a phone's screenshot before the agent ever sees the real one [1][2].
- CSA assesses the most consequential chain in this research as one requiring no root access or system-level privilege: a malicious app holding only the ordinary overlay permission a user grants once through Android Settings hides an instruction on screen, the agent's VLM reads it as legitimate user intent, and the agent – running on a host PC that drives the phone over ADB – passes the resulting text to a shell command with no sanitization, giving the app arbitrary code execution on the controller machine itself [1][2].
- No CVE has been assigned to any of the five frameworks, and the researchers report that private disclosure emails to maintainers have gone unanswered; some of the underlying perception flaws, such as instructions hidden in screen corners or display cutouts, have no complete software-only fix given current hardware constraints [1][2].
- The flaw belongs to a broader, recurring pattern CSA has tracked across the agentic AI ecosystem: agents that cannot reliably separate attacker-supplied perceptual content from legitimate operator instruction, whether that content arrives as an image, an error log, or – as here – a few invisible pixels on a phone screen (see CSA Resource Alignment, below).

Background

Mobile AI agents automate Android devices by closing a perceive-act loop: a controller, typically running on a host PC, requests a screenshot from the phone over the Android Debug Bridge (ADB), feeds that image to a vision-language model, and translates the model's interpretation of what it sees into the next action – a tap, a swipe, or a shell command issued back to the device. This architecture has become

popular for tasks ranging from app testing and accessibility automation to agentic assistants that book appointments or manage messages on a user's behalf, and a cluster of open-source frameworks – AppAgent, AppAgentX, Mobile-Agent, Open-AutoGLM, and MobA among them – has grown up around it, each pairing a different underlying VLM with its own screenshot-and-command pipeline [2].

That architecture assumes the phone's screen shows the agent an honest picture of device state and that any other app sharing the device is a passive bystander. Security researchers from Simon Fraser University, the Chinese University of Hong Kong, Shandong University, and Xingtu Lab at Qi An Xin (QAX) tested that assumption directly, publishing their findings in a paper titled "(A)I Sees What You Don't: Exploiting New Attack Surfaces in Third-Party Mobile Agents," posted to arXiv on July 1, 2026, and revised July 14 [2]. The Hacker News reported on the disclosure on July 21, framing the core finding succinctly: invisible text on an Android screen can hijack an AI agent and make it run commands on the PC controlling the phone [1]. The researchers organize their findings around two attack surfaces they argue are structural to this class of agent rather than incidental bugs in any one implementation: a screen perception attack surface, which exploits the gap between what a human eye and a machine vision model each register from the same image, and a misused channel attack surface, which targets the plumbing that carries screenshots and commands between phone and controller [2].

To demonstrate both, the researchers built seven distinct attack techniques and ran each against all five frameworks, using a background Android app that needs only two Android permissions: `SYSTEM_ALERT_WINDOW`, the special-access overlay permission a user grants once through Settings, and ordinary shared-storage write access [2]. Every framework proved vulnerable to at least six of the seven techniques, and the researchers were able to chain several of them into complete, end-to-end compromises reaching from an ordinary app installed on the phone to arbitrary command execution on the host machine that controls it [1][2].

Security Analysis

Text no human eye can see, but every model reads

The foundation of the attack is what the researchers call subliminal visual injection: rendering instructions as an overlay at opacities as low as 2%, a contrast level far below what a human viewer notices on a normal screen but well within the range a VLM's image encoder resolves as legible text [2]. Tested across opacity levels from 2% to 20%, every model the researchers evaluated extracted the hidden instruction in the overwhelming majority of trials, including the two frontier closed models most commonly deployed in production agent pipelines and a lightweight nine-billion-parameter open model, which suggests the vulnerability is not a quirk of any single model's training but a shared property of how

VLMs process visual input at low contrast [2]. A related technique places payloads inside a device's rounded screen corners or display cutout – areas a human simply does not look at but that remain fully visible to software reading the raw frame buffer – and a third uses chrominance steganography, altering only the color channels of an image while preserving luminance so the change is imperceptible to a viewer but recoverable by a model [2]. The researchers were blunt about the limits of a fix: for corner- and cutout-based injection in particular, they found "no straightforward and effective software-based solution," since the invisibility depends on hardware display geometry rather than any adjustable software setting [2].

A race between the real screenshot and the fake one

A second technique attacks the plumbing rather than the perception model directly. Because these frameworks write each screenshot to a predictable, shared-storage path – `/sdcard/screenshot.png` in one framework, `/sdcard/tmp.png` in another – and because a measurable window opens between the moment the device finishes capturing the frame and the moment the host controller retrieves it, a malicious background app can detect that window and substitute its own doctored image before the real one is read. The researchers measured this time-of-check-to-time-of-use (TOCTOU) window at 50 to 500 milliseconds across the tested frameworks, averaging roughly 210 milliseconds over 100 trials, and demonstrated that a background service polling every 5 to 10 milliseconds has sufficient time to take a file lock signaling that capture has completed, overwrite the image, and release the lock before the controller pulls the file [2]. Executed against four of the five frameworks, the substitution succeeded in 19 or 20 of 20 attempts; the fifth, MobA, proved immune because it streams frames directly from the device rather than staging them in shared storage, which the researchers flag as the one architectural choice among the five that meaningfully closes this specific gap [2].

From a phone screen to a shell on the controller PC

CSA regards this step as the chain's most consequential, since it turns a perception flaw on a phone into code execution on a completely different machine. In four of the five frameworks, once the VLM has been made to output attacker-chosen text – whether through subliminal injection or a tampered screenshot – that text flows, unsanitized, into a call resembling `subprocess.run(adb_command, shell=True)` on the host controller. Because `shell=True` hands the string to a system shell for interpretation rather than executing it as a literal argument list, ordinary shell metacharacters embedded in the model's output are interpreted as command separators. The researchers first demonstrated the underlying mechanism in a one-off proof-of-concept against AppAgent, using a real-world WeChat case study in which a payload resembling `test;pwd>rce_success` showed that the semicolon

terminates the intended ADB command and launches an arbitrary second command chosen by the attacker. In a separate, standardized evaluation of the same technique across all four vulnerable frameworks, they used a `calc.exe`-triggering payload instead and achieved 20-of-20 success, launching a calculator application on the host operating system each time to make the code execution visible [2]. Open-AutoGLM was the one framework immune to this specific technique, because it base64-encodes agent output before it reaches the shell, neutralizing the metacharacters that make the injection work – a narrow counterexample showing the flaw is an implementation choice, not an unavoidable property of the architecture [2].

The remaining techniques: spoofed interfaces, sniffed input, and open broadcasts

The paper documents three further techniques that round out the seven-attack suite: a UI spoofing overlay that mimics a login screen so convincingly that an agent, following its normal task flow, enters a user's real password into the attacker's fake field; an Accessibility Service listener that captures plaintext input, including passwords, by monitoring standard Android text-change events; and interception of an unprotected broadcast channel that some frameworks use to relay typed text between the controller and the device, which a malicious receiver can read without holding any special permission [2]. The table below summarizes all seven techniques and where each one succeeded.

Attack	Mechanism	Frameworks defeated
Subliminal visual injection	Overlay text at 2-20% opacity, invisible to humans, legible to VLMs	All 5 (separately, VLMs read the hidden text in 18-20 of 20 trials – a model-level detection rate, not a framework-compromise count)
Invisible zone injection	Payload hidden in rounded corners / display cutout regions	All 5 (no full software fix identified)
UI spoofing	Overlay mimics a login screen to capture credentials	All 5
Screenshot tampering (TOCTOU)	File-lock polling to swap the screenshot in a 50-500ms write/read gap	4 of 5 (MobA immune via memory-only streaming)
Broadcast input interception	Unprotected broadcast channel relays typed text in the clear	3 of 5 (AppAgent, AppAgentX immune – they use native <code>adb</code>)

Attack	Mechanism	Frameworks defeated
		<code>shell input text</code> rather than a broadcast channel)
Credential sniffing	Accessibility Service reads plaintext password-entry events	All 5
Host-side command injection	Unsanitized model output reaches <code>subprocess.run(..., shell=True)</code>	4 of 5 (Open-AutoGLM immune via base64 encoding)

No CVE identifier has been assigned to any of the five frameworks as of this writing, and the researchers report that the coordinated-disclosure emails they sent to maintainers have so far gone unanswered [1] [2]. Taken together, the seven techniques indicate that the compromise of a mobile AI agent does not require compromising the phone's operating system, the agent's model weights, or the network channel between phone and controller – an ordinary, low-privilege app sharing the same device is sufficient.

Recommendations

Immediate Actions

Organizations piloting or operating any of the five affected frameworks – AppAgent, AppAgentX, Mobile-Agent v3, Open-AutoGLM, or MobA – should treat them as unpatched pending upstream response and audit their deployment for the two highest-severity gaps: any controller code path that passes model-derived text into a shell with `shell=True` or equivalent string concatenation, and any screenshot pipeline that stages frames in world-readable shared storage rather than streaming them directly. Any organization running these frameworks against a device with production credentials, personal accounts, or payment access should suspend that use until the command-injection path is closed, since the researchers demonstrated arbitrary code execution on the controller machine, not merely the phone.

Short-Term Mitigations

Teams that must continue operating these frameworks should replace shell-string execution with argument-list subprocess calls that never invoke a shell interpreter, adopt memory-only or `exec-out` -style screenshot streaming to eliminate the TOCTOU window entirely, and apply signature-level permissions to any broadcast channel used to relay input between controller and device. Contrast-enhancement preprocessing before an image reaches the VLM offers partial protection against low-opacity overlay injection but does not address corner, cutout, or chrominance-based techniques, so it should be treated as one layer in a defense-in-depth posture rather than a complete fix. Maintaining a per-task allowlist of foreground applications and diffing it against the actual foreground activity before each agent action can catch UI-spoofing attempts that the perception layer alone will miss.

Strategic Considerations

This disclosure extends a pattern CSA has now documented across several distinct agent modalities (see CSA Resource Alignment, below): agents that perceive their environment through a channel they cannot cryptographically verify – an image, an error log, a calendar invitation, a contact card – will act on attacker content injected into that channel as though it were legitimate instruction, because the model has no architectural basis for telling the two apart. Enterprises evaluating mobile automation or VLM-driven testing tools should extend the same vendor-risk scrutiny they apply to third-party code to the perception pipeline itself, asking specifically whether a vendor's agent trusts screen content from any co-resident application and whether model output ever reaches a shell without passing through an allowlist or argument-list boundary. Longer term, the corner- and cutout-injection findings argue for treating perception-layer integrity as a hardware-and-software co-design problem, not something a downstream software patch alone can close.

CSA Resource Alignment

This disclosure is a direct instance of the image-based prompt injection pattern CSA's AI Safety Initiative documented in "[Image-Based Prompt Injection: Hijacking Multimodal LLMs Through Visually Embedded Adversarial Instructions](#)," which analyzed typographic, steganographic, and adversarial-perturbation techniques for embedding attacker instructions in images that a vision-language model reads but a human does not [3]. The mobile agent research adds a new, physically-grounded variant to that taxonomy – instructions hidden in screen opacity, display cutouts, and color-channel steganography – and confirms the same recommendation that research made: images and screenshots from any source an agent does not fully control must be treated with the same skepticism as untrusted text input, not as

passively trustworthy sensor data. The command-injection finding, in which a compromised agent executes attacker commands using the developer's or operator's own system privileges, matches the structural pattern CSA described in "[Confused Deputy Attacks on Autonomous AI Agents](#)": an agent unable to verify the provenance of an instruction becomes a privileged actor manipulated into serving an attacker's goals while operating on credentials that belong to its legitimate operator [4]. Both findings map cleanly onto the Agent Frameworks and Deployment Infrastructure layers of CSA's "[MAESTRO](#)" agentic AI threat modeling framework, which was built specifically to decompose vulnerabilities like these – perception manipulation, trust-boundary confusion between data and instruction, and unsandboxed action execution – layer by layer rather than treating them as one-off implementation bugs [5]. Organizations formalizing controls against this attack class should anchor them in the AI Controls Matrix (AICM v1.1), whose Application and Interface Security domain is oriented toward input validation at AI system boundaries, complementing controls elsewhere in the matrix that address least-privilege execution for AI agents acting on a user's or developer's behalf [6].

References

- [1] The Hacker News. "[Open-Source Android AI Agents Could Let Invisible Screen Text Run Code on Host PCs.](#)" The Hacker News, July 21, 2026.
- [2] Zhang, Z., Xie, Z., Diao, W., and Wu, J. "[\(A\)I Sees What You Don't: Exploiting New Attack Surfaces in Third-Party Mobile Agents.](#)" arXiv, July 1, 2026 (revised July 14, 2026).
- [3] Cloud Security Alliance AI Safety Initiative. "[Image-Based Prompt Injection: Hijacking Multimodal LLMs Through Visually Embedded Adversarial Instructions.](#)" CSA Lab Space, March 8, 2026.
- [4] Cloud Security Alliance AI Safety Initiative. "[Confused Deputy Attacks on Autonomous AI Agents.](#)" CSA Lab Space, March 23, 2026.
- [5] Cloud Security Alliance. "[Agentic AI Threat Modeling Framework: MAESTRO.](#)" CSA Blog, February 6, 2025.
- [6] Cloud Security Alliance. "[AI Controls Matrix \(AICM\) v1.1.](#)" Cloud Security Alliance, 2026.