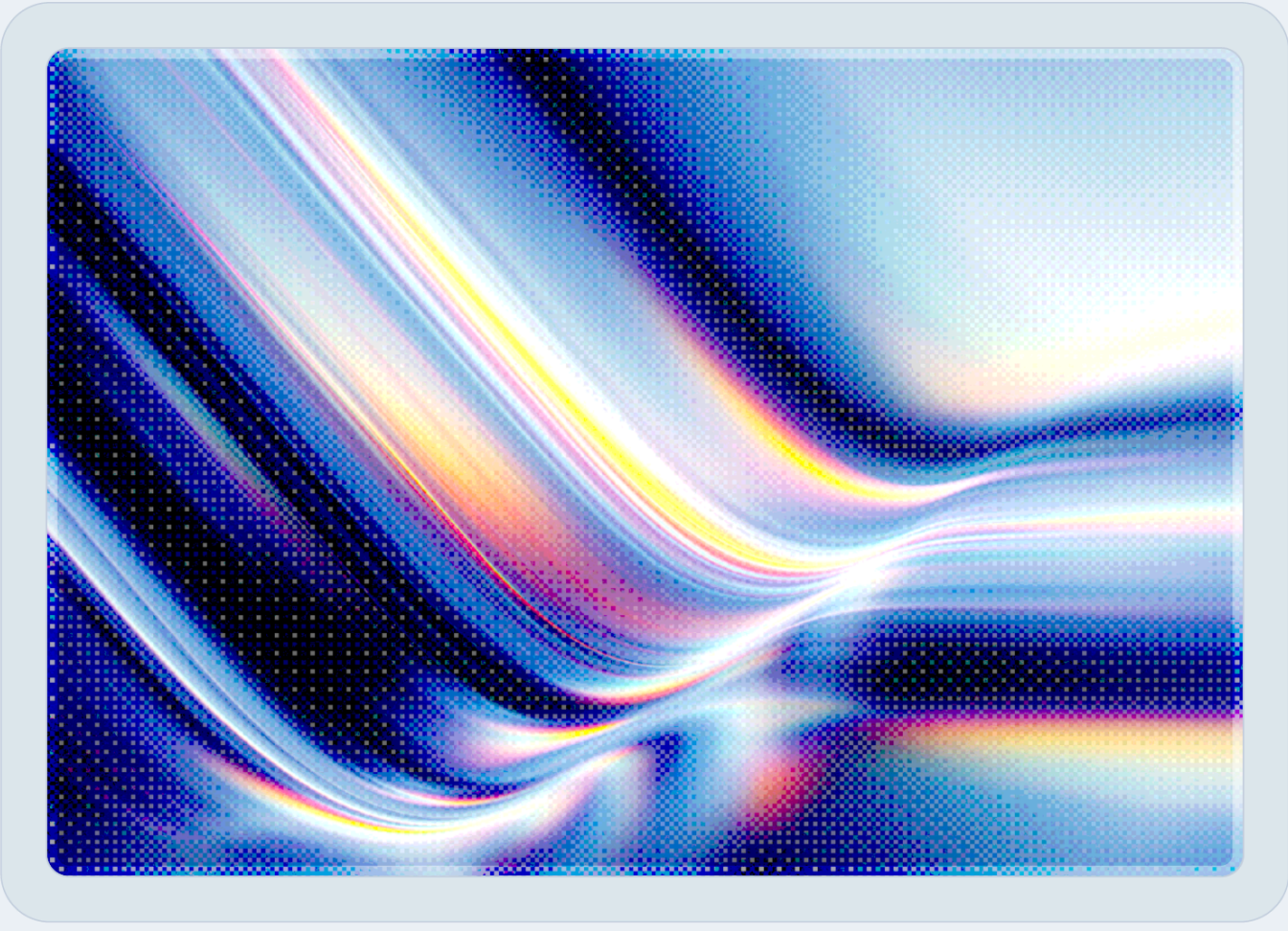
CSAI Foundation | Cloud Security Alliance

Invisible Screen Text Hijacks Android AI Agents

Screen-Perception Flaws Let Malicious Apps Run Code on the Host PC

2026-08-03

AI-assisted Rapid Research



CSAI

CSA cloud security alliance®

Key Takeaways

Academic researchers have shown that five widely used open-source Android AI agent frameworks can be hijacked through text a human operator never sees. By drawing instructions onto the screen at roughly 2% opacity, a co-installed malicious app can plant a payload that survives a routine screenshot: invisible to the person watching the phone, but fully legible to the vision-language model reading the same image. Every framework tested failed at least six of seven distinct attack variants, and in the most severe case the injected text reached an unsanitized shell command on the PC controlling the device, giving an attacker arbitrary code execution on that host [1][2]. The underlying frameworks have collectively drawn tens of thousands of GitHub stars, and the researchers report that maintainers did not respond to private disclosure before publication, leaving the vulnerable code on public main branches [1] [2]. No CVE identifiers have been assigned, and there is no evidence of in-the-wild exploitation to date. This is consistent with a durable architectural weakness rather than a single patchable bug: the same failure pattern — trusted and untrusted content sharing an unauthenticated channel — recurs across CSA's prior work on agentic AI attack surfaces [3][4][5].

Background

Mobile AI agents built on vision-language models have moved from research demonstrations to open-source tools with substantial followings — the five frameworks examined in this note have collectively drawn tens of thousands of GitHub stars. Frameworks such as AppAgent, AppAgentX, Mobile-Agent-v3, Open-AutoGLM, and MobA give a model control of an Android device by capturing periodic screenshots, interpreting what is on screen, and issuing the corresponding taps, swipes, and shell commands back to the device, typically over the Android Debug Bridge (ADB) from a controlling PC. This design pattern trades the fixed, script-based automation of earlier mobile tooling for something closer to a human operator: the agent looks at the screen and decides what to do next, the same way a person would. That flexibility is also the design's exposure. A model that "reads" the screen to decide its next action has no independent way to confirm that everything visible on that screen was placed there by a legitimate, foreground application rather than by an unrelated app exploiting Android's shared display and storage surfaces.

A paper published to arXiv on July 1, 2026, and revised July 14, 2026 — "(A)I Sees What You Don't: Exploiting New Attack Surfaces in Third-Party Mobile Agents," by Zidong Zhang, Zhentao Xie, Wenrui Diao, and Jianliang Wu — formalizes this exposure into two attack surfaces [2]. The first, which the authors call the Screen Perception Attack Surface, exploits the gap between what a human eye can register on a phone display and what a vision-language model can extract from the same raster image; a payload rendered at low enough opacity, or placed in a region of the frame buffer the display hardware does not show, is effectively invisible to a person but remains fully present in the pixel data the agent processes. The second, the Misused Channel Attack Surface, targets the plumbing between perception and action — the screenshot files, ADB broadcasts, and accessibility-service hooks the agent relies on to move information from the device to the model and commands back again. The Hacker News reported on the findings on July 21, 2026, noting that Open-AutoGLM alone carries more than 25,000 GitHub stars, an indication of adoption well beyond isolated research prototypes [1].

Security Analysis

By the researchers' own account, the most severe of the seven attack variants is a command injection chain that begins with the invisible overlay. A malicious Android application with only ordinary, non-privileged permissions — the ability to draw over other windows and write to shared external storage — renders text at approximately 2% opacity, a level low enough that human testers could not perceive it on the physical device across repeated trials. When the AI agent captures its next screenshot to decide on an action, that invisible text is captured along with everything else, and the vision-language model reads it as if it were legitimate on-screen content. The researchers tested six vision-language models, including GPT-4o, Claude Opus 4.5, Gemini 3 Pro, and GLM-4V, each of which detected and transcribed the low-opacity payload in 20 of 20 trials; the weakest performer, the domain-specific AutoGLM-Phone, still detected it in 18 of 20, confirming that the perceptual gap the attack depends on is real and consistent across both general-purpose and domain-specific models [2].

The chain becomes exploitable because several of the tested frameworks pass model-generated text directly into a shell rather than treating it as untrusted input. AppAgent's implementation, for example, constructs ADB commands using Python's `subprocess.run()` with `shell=True`, meaning any shell metacharacters embedded in the agent's output are interpreted by the host operating system rather than passed as inert arguments. Because the invisible on-screen text becomes part of the model's context and, in turn, part of the text the model emits when deciding how to act, an attacker who controls the injected text can effectively control the string that reaches the shell. In their proof-of-concept, the researchers used this path to launch a benign target process (`calc.exe`) on the Windows host

controlling the phone in 20 of 20 trials across four of the five frameworks, demonstrating reliable, repeatable remote code execution on the operator's own machine rather than merely on the sandboxed Android device [1].

The remaining five measured attack variants target weaknesses in how these frameworks handle the screenshot-and-broadcast pipeline itself. Screenshot files are frequently written to unprotected, world-readable locations such as `/sdcard/`, and the researchers measured a window of roughly 50 to 500 milliseconds between when a screenshot is captured and when the agent retrieves it — long enough for a race-condition attack to swap or tamper with the file before it is read. Other variants include fake overlay UI elements that display spoofed system messages (for instance, a bogus "network sync in progress" prompt designed to stall the agent or extract a wait period useful to the attacker), injection into bezel or display-cutout regions of the frame buffer that exist in the raw image but are never rendered on the physical screen, fake login prompts paired with accessibility-service hooks to harvest credentials the agent enters on the user's behalf, and interception of unprotected ADB input broadcasts, which on several frameworks can be received by any application installed on the device rather than being scoped to the agent itself [2]. The paper's seventh variant — embedding instructions in the chrominance channel of a captured image rather than manipulating overlay opacity — is presented as a theoretical extension of the perception-gap concept; the researchers describe it but did not empirically test it against the five frameworks [2].

A recurring theme across the paper's findings is that confirmation prompts and other in-model safeguards do not meaningfully close this gap. Open-AutoGLM, for instance, asks the model to confirm before taking sensitive actions, but that confirmation step is itself mediated by the same compromised perception pipeline — if the model has already been deceived about what is on screen, asking it to double-check offers no independent verification. The researchers frame this succinctly: the language model is not a security boundary. Compounding the exposure, Open-AutoGLM's own setup documentation — attached to a repository with more than 25,000 GitHub stars — walks new users through enabling USB debugging and installing a test keyboard, recreating nearly every precondition the attack requires short of the malicious app itself [1].

The researchers privately contacted the maintainers of the affected projects ahead of publication but received no response from any of them. As of July 17, 2026, the vulnerable code remained on the public main branches of all five repositories, and none of the projects maintain a formal security policy or a dedicated vulnerability-reporting channel, which the paper identifies as a structural gap in how fast-moving open-source AI agent tooling handles responsible disclosure [1][2]. No CVE identifiers have been assigned to any of the seven attack variants, and the researchers report no evidence that the techniques have been used against real deployments to date. That absence of known exploitation should not be read as an absence of risk: the attack requires no elevated Android permissions, works against a co-

installed unprivileged app, and — for the shell-injection variant — produces reliable code execution on the host machine an operator uses to control the phone, which in enterprise contexts is frequently the same machine holding credentials, VPN access, and other sensitive resources.

Recommendations

Immediate Actions

Organizations that have deployed, or are piloting, any of the affected open-source frameworks — AppAgent, AppAgentX, Mobile-Agent-v3, Open-AutoGLM, or MobA — should treat them as unpatched pending upstream fixes and restrict their use to isolated, disposable test devices and hosts with no access to production credentials or internal networks. Any invocation of `subprocess.run(..., shell=True)` or equivalent shell-interpolation patterns in agent code should be replaced with argument-list execution that does not interpret metacharacters, eliminating the specific path the researchers used to achieve code execution. Screenshot retrieval workflows that rely on world-readable storage locations should be audited, and where feasible, screenshots should be streamed directly to the controlling process over a secured channel rather than written to a shared filesystem location where a race condition can be exploited.

Short-Term Mitigations

Security teams operating mobile AI agents should implement signature-level permission scoping on ADB input broadcasts so that only the agent's own controller process can send or receive them, closing the broadcast-interception variant. Agents should also monitor for unexpected foreground activity changes during a task and maintain a per-task allowlist of expected application packages, flagging or halting execution when an unanticipated app comes into the foreground during an automated session. Where the underlying vision-language model or capture pipeline supports it, applying contrast enhancement or a fixed luminance floor to captured screenshots before they reach the model can degrade the reliability of low-opacity overlay payloads, though the researchers note this is a partial mitigation rather than a structural fix, and it does nothing against bezel or cutout injection, which occurs in frame-buffer regions no display-level filter reaches.

Strategic Considerations

The durable fix is architectural rather than a patch to any single framework: agents that act on what they perceive from a shared, multi-tenant display surface need a way to distinguish content that originated from the foreground application the agent believes it is controlling from content injected by any other co-resident process. This is the same structural problem underlying indirect prompt injection more broadly — attacker-controlled content masquerading as trusted context that the agent has no architectural means to authenticate — regardless of whether that content arrives as screen pixels, document metadata, or tool output. Enterprises building governance programs around agentic AI should treat "does this agent have a verifiable way to distinguish trusted display or data content from untrusted content sharing the same channel" as a standing procurement and architecture-review question, not a one-time assessment tied to this specific disclosure.

CSA Resource Alignment

CSA's Agentic AI Red Teaming Guide already anticipates this general failure class. The guide instructs red teamers to carry application-security fundamentals — including input validation and authentication/authorization testing — directly into agent evaluation, and it names "Agent Frameworks: Mitigating backdoor attacks, input validation exploits, and supply chain vulnerabilities" as a distinct testing category. The Android agent findings are a concrete instance of that category: model output reaching an unsanitized shell call is an input-validation failure of a kind the guide already tells practitioners to probe for, here reached through a screen-perception channel rather than a text prompt.

CSA's MAESTRO agentic AI threat modeling framework decomposes agent security into seven layers, and this disclosure spans several of them simultaneously: the Agent Frameworks layer (unvalidated command construction in AppAgent and comparable tools), the Deployment and Infrastructure layer (unprotected screenshot storage and ADB broadcast channels), and the Evaluation and Observability layer (the absence of any check for anomalous or low-opacity content in captured screen data). Threat modeling exercises for mobile or desktop screen-perception agents should walk all seven MAESTRO layers rather than treating the vision pipeline as a black box outside the framework's scope.

CSA's guidance on using Zero Trust to secure enterprise information in LLM environments calls for agentic AI deployments to operate under tightly scoped, role-based, action-specific permissions and continuous, auditable monitoring of agent actions and cross-system data flows. That guidance is consistent with the mitigations recommended above: signature-level permission scoping on ADB broadcasts and per-task package allowlisting are Zero-Trust least-privilege and continuous-monitoring controls applied to the specific channels this research shows are otherwise wide open.

Organizations evaluating mobile AI agent deployments should also map this disclosure against the AI Controls Matrix (AICM) v1.1, particularly its Application and Interface Security and Threat and Vulnerability Management domains, which cover both the command-injection and input-validation failure modes this research identifies [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, Zidong; Xie, Zhentao; Diao, Wenrui; Wu, Jianliang. "(A)I Sees What You Don't: Exploiting New Attack Surfaces in Third-Party Mobile Agents." arXiv:2607.00333, July 1, 2026 (revised July 14, 2026).

[3] Cloud Security Alliance. "Agentic AI Red Teaming Guide." Cloud Security Alliance, May 28, 2025.

[4] Cloud Security Alliance. "Agentic AI Threat Modeling Framework: MAESTRO." Cloud Security Alliance, February 6, 2025.

[5] Cloud Security Alliance. "Using Zero Trust to Secure Enterprise Information in LLM Environments." Cloud Security Alliance, March 2, 2026.

[6] Cloud Security Alliance. "AI Controls Matrix (AICM) v1.1." Cloud Security Alliance, 2026.

