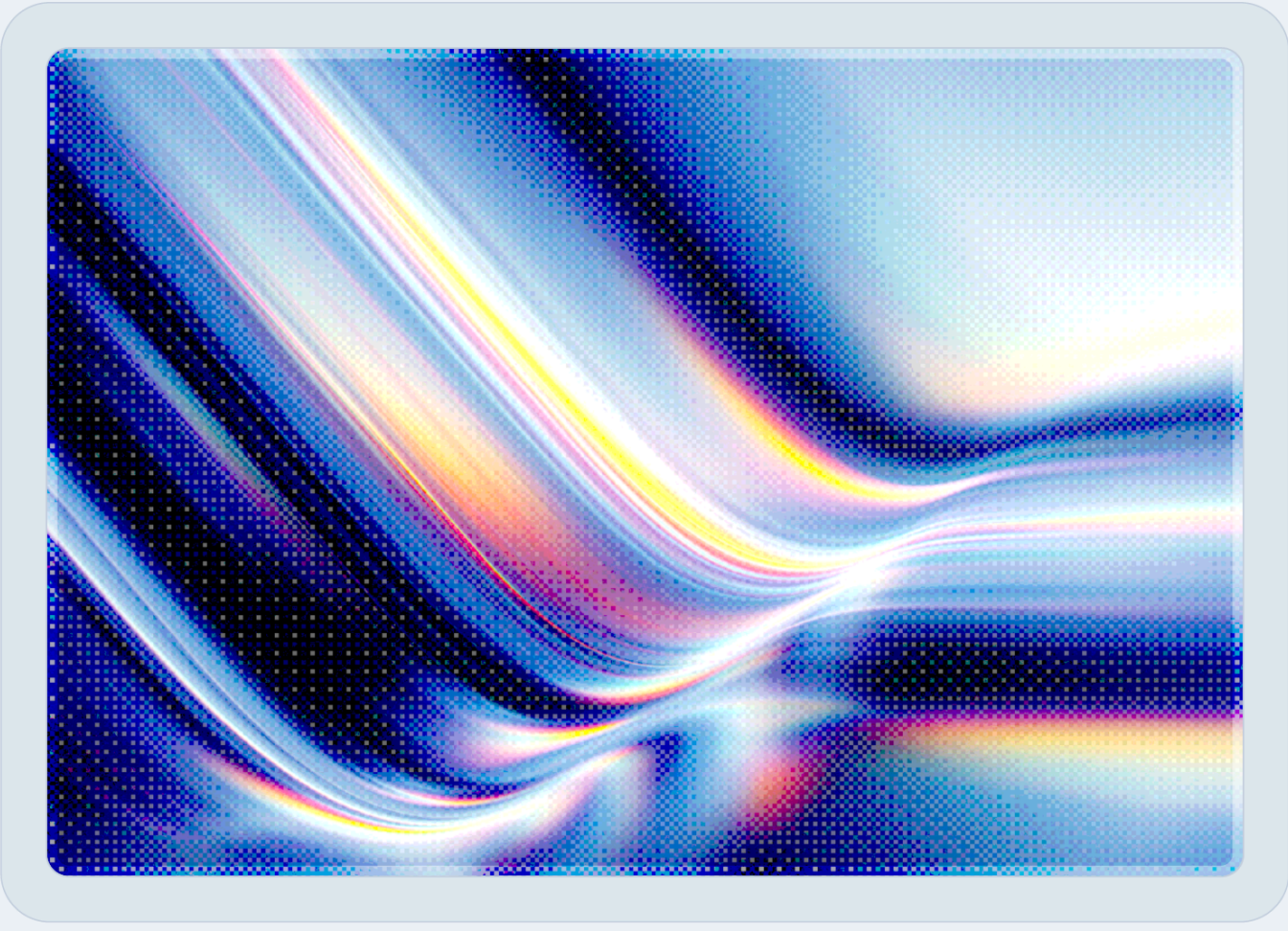
CSAI Foundation | Cloud Security Alliance

RatHat: A Live AI Agent Controls Compromised Android Devices

2026-09-22

AI-assisted Rapid Research



CSAI

CSA cloud security alliance®

Key Takeaways

- Zimperium zLabs researchers Gianluca Braga, Vishnu Pratapagiri, and Fernando Ortega disclosed RatHat on September 16, 2026, an Android trojan that serializes the device's Accessibility tree into XML and sends it to a generative AI assistant, which returns coordinates and navigation commands such as "SCROLL_DOWN" to drive the phone in place of a hard-coded automation script [1][2].
- The malware abuses Android's Accessibility Service to silently enable Developer Options and Wireless Debugging, read the six-digit ADB pairing code off the screen, and self-pair with the device's own debug interface, giving the attacker a local shell without ever connecting an external computer — a technique also seen in the ToxicPanda and RedHook malware families [1].
- Two Go-language components carry out the attack: a service disguised as the native library `liblocal-service.so` executes shell commands over that self-paired ADB session, while `libmedia_codec.so` runs an FRP (Fast Reverse Proxy) client that holds open a persistent tunnel back to attacker infrastructure [1][2].
- RatHat intercepts the uninstall workflow, cancels removal attempts, and displays a spoofed Google Play error; because the Go agent independently retains ADB shell access, a device can remain compromised even after the visible application is removed, and Zimperium and follow-on reporting note that a factory reset is generally required to fully clear it [2][4].
- Credential theft combines banking- and cryptocurrency-app overlays, SMS and OTP interception, browser URL scraping, and a hardware-level keylogger that reads raw touch coordinates from `/dev/input` and maps them to brand-specific keypad layouts to reconstruct PINs and unlock patterns, with overlays and interception also observed against WeChat and Alipay [3].
- Zimperium assesses the malware is linked to China-based threat actors based on Chinese-language prompts embedded in the code that drive the AI-automation subsystem, and distribution runs through malvertising, SMS phishing (smishing), and phishing sites offering APK downloads outside Google Play [1][3].

Background

RatHat surfaced publicly on September 16, 2026, when Zimperium's zLabs mobile threat research team published a technical analysis describing an Android remote access trojan whose defining feature is delegation of real-time interface navigation to a generative AI assistant rather than to fixed, pre-scripted automation routines [1]. Conventional Android banking trojans and remote access tools typically drive the user interface through hard-coded UI automation: a script identifies a button by its resource ID or fixed screen coordinates and taps it, a method that breaks whenever a targeted app updates its layout or a device runs a different screen resolution. RatHat instead captures the live Accessibility tree — the structured representation of every visible interface element that Android exposes to assistive technologies — serializes it to XML, and transmits it to an AI assistant that Zimperium's report does not name. The model returns instructions describing where to tap, what text is present on screen, and when to scroll, which the malware executes as synthetic input events. Zimperium researchers characterized the AI's role as resolving a target element's center coordinates as JSON to direct synthetic clicks, a narrow and largely mechanical task, but one that gives the malware's navigation logic the adaptability of a human operator without requiring a human to be present for each interaction [3].

That adaptability compounds a set of privilege-escalation and persistence techniques that are individually familiar from prior Android malware families but that RatHat assembles into an unusually complete chain. Like ToxicPanda and RedHook before it, RatHat abuses granted Accessibility permissions to navigate into the device's Developer Options menu, enable Wireless Debugging, and extract the ADB pairing code that Android displays during that process, all without the user's awareness [1]. Because ADB (Android Debug Bridge) is designed to be paired locally over the device's own Wi-Fi connection rather than exclusively through a wired connection to a trusted computer, RatHat can complete this self-pairing entirely on-device and obtain a functioning shell with elevated command execution rights. That shell access is then handed to a native-looking Go binary that persists independently of the visible application, which is the mechanism Zimperium and subsequent reporting cite as the reason RatHat can survive an apparent uninstall [1][2][4].

Distribution follows patterns well established in the Android malware ecosystem: malvertising campaigns, SMS phishing messages, and deceptive third-party download portals direct victims to pages that mimic legitimate app stores or trusted brands and prompt a sideloaded APK install [1][3][4]. Google told BleepingComputer that no application containing RatHat has been identified on Google Play and that Google Play Protect automatically blocks known versions of the malware, indicating the observed campaigns rely on sideloading outside the official store rather than a Play Store distribution failure [1]. Zimperium's attribution assessment — that the operators are China-based — rests on Chinese-language

text found in the prompts the malware sends to its AI assistant, a signal that speaks to the developers' likely origin without establishing which victim populations or regions the campaign is currently targeting [1].

Security Analysis

RatHat's architecture separates concerns across three layers. The malicious Android application itself functions primarily as a permission-acquisition and dropper layer: it social-engineers the victim into granting Accessibility Service access, the permission from which nearly all of RatHat's subsequent capability appears to flow. From there, the Go agent disguised as `liblocal-service.so` takes over command execution, using the self-paired local ADB shell to run commands with elevated privileges the base application could not otherwise obtain, and separately checks whether the visible application has been removed and silently reinstalls it if so [1][2]. The second Go component, packaged as `libmedia_codec.so`, functions as an FRP reverse-proxy client, establishing an outbound tunnel that gives the attacker's command-and-control infrastructure an addressable path back into the device even though the phone itself sits behind carrier-grade NAT or a home router [1]. Security Affairs further reports that RatHat maintains multiple simultaneous C2 channels — HTTP, WebSocket, and the FRP tunnel — which gives the operator redundancy if any single channel is blocked or detected [3].

The credential-theft toolkit combines several individually familiar techniques rather than relying on any single novel method. RatHat displays HTML overlay screens on top of targeted banking and cryptocurrency applications to capture entered credentials directly — with overlays and interception also observed against the messaging and payment apps WeChat and Alipay [3] — and separately intercepts SMS messages and notifications to capture one-time passwords and multi-factor authentication codes before the legitimate app can process them [1][3][4]. RatHat extends beyond Accessibility-based text-field capture, which some banking apps defeat by disabling Accessibility rendering on sensitive screens, with hardware-level keylogging: it reads raw touch-event coordinates directly from the `/dev/input` device driver and cross-references them against a database of brand-specific keypad layouts to reconstruct PINs and unlock patterns even when it cannot see which on-screen element was tapped [3][4]. Combined with browser URL scraping, this gives the malware multiple independent paths to the same credential, so that a defense effective against one collection method does not necessarily block the others.

Capability	Mechanism	Primary Component
UI navigation	Accessibility tree serialized to XML, sent to AI assistant for coordinate/text resolution	Base application + AI assistant API
Privilege escalation	Silently enables Developer Options, Wireless Debugging, self-pairs local ADB	Base application (Accessibility abuse)
Command execution	Shell commands over self-paired ADB session	`liblocal-service.so` (Go agent)
Remote access	Persistent reverse-proxy tunnel to C2	`libmedia_codec.so` (FRP client)
Credential theft	Banking/crypto/WeChat/Alipay overlays; SMS and OTP interception; browser URL scraping	Base application
PIN/pattern theft	Raw touch coordinates from `/dev/input` matched to keypad layouts	Hardware-level keylogger
Persistence	Cancels uninstall, spoofs Google Play error, silently reinstalls APK	Go agent + base application
Anti-analysis	XOR/subtraction and StringFog string encryption, 61 MB manifest, invalid DEX pseudo-instructions, debugger/emulator/Frida/Xposed detection [1] [3]	Base application

RatHat's anti-analysis layer is built to frustrate both automated scanning and manual reverse engineering. Reported techniques include encrypting internal strings with a combination of simple XOR, byte-subtraction, and the StringFog obfuscation library; bloating the AndroidManifest.xml to roughly 61 megabytes with unusual data blocks; inserting invalid Dalvik bytecode pseudo-instructions designed to break disassemblers; and modifying the APK's ZIP container structure in ways that some analysis tools fail to parse correctly [1][3]. The application also performs runtime checks for debuggers, `ptrace` attachment, the Frida instrumentation framework, Xposed hooking, and common emulator fingerprints, and uses Android's SessionInstaller APIs to bypass certain installation restrictions during the drop phase

[3]. None of these individual techniques is unprecedented in Android malware, but in the CSA research team's assessment their density in a single sample suggests a more resourced operation than a typical commodity trojan represents.

The persistence mechanism departs from standard incident-response guidance, which typically assumes that uninstalling the offending application removes the threat. RatHat is specifically engineered to defeat that assumption: when a victim attempts to uninstall the visible app, RatHat's Accessibility-based monitoring intercepts the confirmation dialog, cancels the removal, and displays a fabricated Google Play error message suggesting the uninstall failed for an unrelated reason [1]. Even where a user succeeds in removing the visible application through alternative means, the independently running Go agent retains its ADB shell access and can silently reinstall the APK and restore the Accessibility and Developer Options permissions it depends on [2]. Malwarebytes recommends a factory reset as the reliable remediation once a device is confirmed compromised, rather than a standard app removal [4].

Recommendations

Immediate Actions

- Treat any request to enable Accessibility Service permissions from an application obtained outside Google Play — or from any application whose Play Store legitimacy has not been independently verified — as a high-confidence compromise indicator, and deny the request.
- Audit mobile fleets for devices with Developer Options or Wireless Debugging enabled that were not deliberately configured by IT or a developer, since RatHat's core privilege-escalation path depends on both being active.
- For any device suspected of RatHat infection, perform a full factory reset rather than a standard application uninstall: the malware's Go-agent component can survive app removal and silently reinstall itself [2][4], and a factory reset is the remediation Malwarebytes recommends for full removal [4].

Short-Term Mitigations

- Extend mobile threat defense (MTD) and enterprise mobility management tooling to alert on outbound FRP (Fast Reverse Proxy) tunnel traffic and unexpected local ADB pairing events, both of which are unusual in normal consumer or enterprise device behavior and map directly to RatHat's persistence and C2 mechanisms.

- Restrict sideloading and enforce installation exclusively from Google Play on managed and BYOD devices with access to corporate banking, email, or authentication apps, since every reported RatHat distribution vector — malvertising, smishing, and third-party download portals — depends on sideloading outside the official store [1][3][4].
- Where feasible, apply behavioral monitoring for anomalous Accessibility Service API usage patterns, such as an app using Accessibility to navigate system settings menus rather than to read or interact with its own interface, which is a stronger indicator of abuse than the mere presence of the permission grant.

Strategic Considerations

- Recognize that AI-directed UI automation substantially reduces the interface-layout brittleness that has historically limited scripted mobile RATs' operational lifespan; malware following this pattern is likely to require detection strategies built around behavioral and network signals — such as the outbound calls to an AI assistant API itself — rather than static UI-automation signatures that assumed a fixed script.
- Incorporate mobile endpoints into the same AI-orchestrated-malware threat model already being applied to desktop and server environments: RatHat is a further data point that runtime dependence on an external AI service is not confined to any single operating system or attacker tier.
- Build tabletop and incident-response exercises around the specific challenge of "reinstall-resistant" persistence, since standard mobile IR playbooks that assume uninstall equals remediation will under-respond to malware families, including RatHat, engineered specifically to defeat that assumption.

CSA Resource Alignment

RatHat fits within a threat pattern CSA's AI-assisted rapid research program has already been tracking: malware that invokes a large language model at runtime to perform tasks a hard-coded script would otherwise handle. CSA's research on this "runtime-LLM malware" track has cataloged families such as PROMPTFLUX and PromptSpy, an Android RAT that similarly calls a generative AI model from within the malware itself, arguing that signature- and IoC-based mobile defenses degrade against this class of threat because the malware's behavior is generated dynamically by the model rather than fixed at build time. RatHat extends that pattern from text generation and command synthesis, the primary use cases that CSA research documents, into live visual UI navigation — the AI assistant is not writing malicious code but interpreting a serialized Accessibility tree and directing synthetic taps in real time, arguably a

more tightly integrated use of an external model than prior runtime-LLM malware, though no comparative detection data across these families is yet available. Defenders following that research's recommendation to reweight detection toward behavioral and process-tree signals, and to inventory outbound calls to AI model-provider APIs by source process, should treat RatHat's AI-assistant traffic as a direct instance of the LLM-interaction surface that guidance advises organizations to monitor.

CSA's research on autonomous agentic AI adversaries documents the broader shift of frontier AI models from passive tools into operational actors within active attack chains, drawing on cases where autonomous agents conducted reconnaissance, exploitation, and command execution with minimal human direction. RatHat's AI-directed interface automation is a smaller-scale, mobile-specific instance of the same underlying transition: rather than a human operator or a static script directing each action, a live model call resolves ambiguity in real time, letting the malware adapt to unfamiliar banking-app layouts the way a human operator would. That research's recommendation to extend enterprise AI-agent threat modeling beyond desktop and cloud environments applies directly here, since RatHat is a further illustration that agentic-style adaptability is neither confined to nation-state cyber-espionage tooling nor to non-mobile platforms.

CSA's "EU Forces Android Open to Rival AI Assistants" separately examined how regulatory changes to the Android AI-assistant ecosystem are expanding the set of applications with legitimate access to screen content, notifications, and accessibility-adjacent functionality, warning that this expansion widens the attack surface available to a malicious or compromised assistant. RatHat is not a rival AI assistant in the regulatory sense that note analyzed, but it exploits the same underlying Android capability — deep Accessibility Service access paired with generative AI-driven decision-making — that the note flagged as a coming governance and vetting challenge for enterprise mobile device management. Organizations building the assistant allow-listing and Accessibility Service scrutiny that note recommends ahead of the EU's compliance deadlines should treat RatHat as a concrete, present-day illustration of the abuse pattern that guidance was written to anticipate, rather than a hypothetical future risk.

All three findings map to the identity and access management and threat and vulnerability management domains of the AI Controls Matrix (AICM v1.1), particularly the controls governing least-privilege permission grants and monitoring of AI model-API interactions, both of which are directly implicated by a malware family whose core capability depends on an over-broad Accessibility Service grant and an unmonitored outbound call to a generative AI endpoint.

References

[1] BleepingComputer. "New RatHat Android malware uses AI to automate device control." BleepingComputer, September 2026.

[2] The Hacker News. "RatHat Android Malware Abuses ADB to Retain Shell Access After Uninstall." The Hacker News, September 2026.

[3] Security Affairs. "RatHat Turns Android Accessibility Into an Attack Weapon." Security Affairs, September 2026.

[4] Malwarebytes. "New Android malware uses AI to steal bank logins and PINs." Malwarebytes Labs, September 2026.

[5] Cloud Security Alliance. "EU Forces Android Open to Rival AI Assistants." Cloud Security Alliance, July 2026.

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

