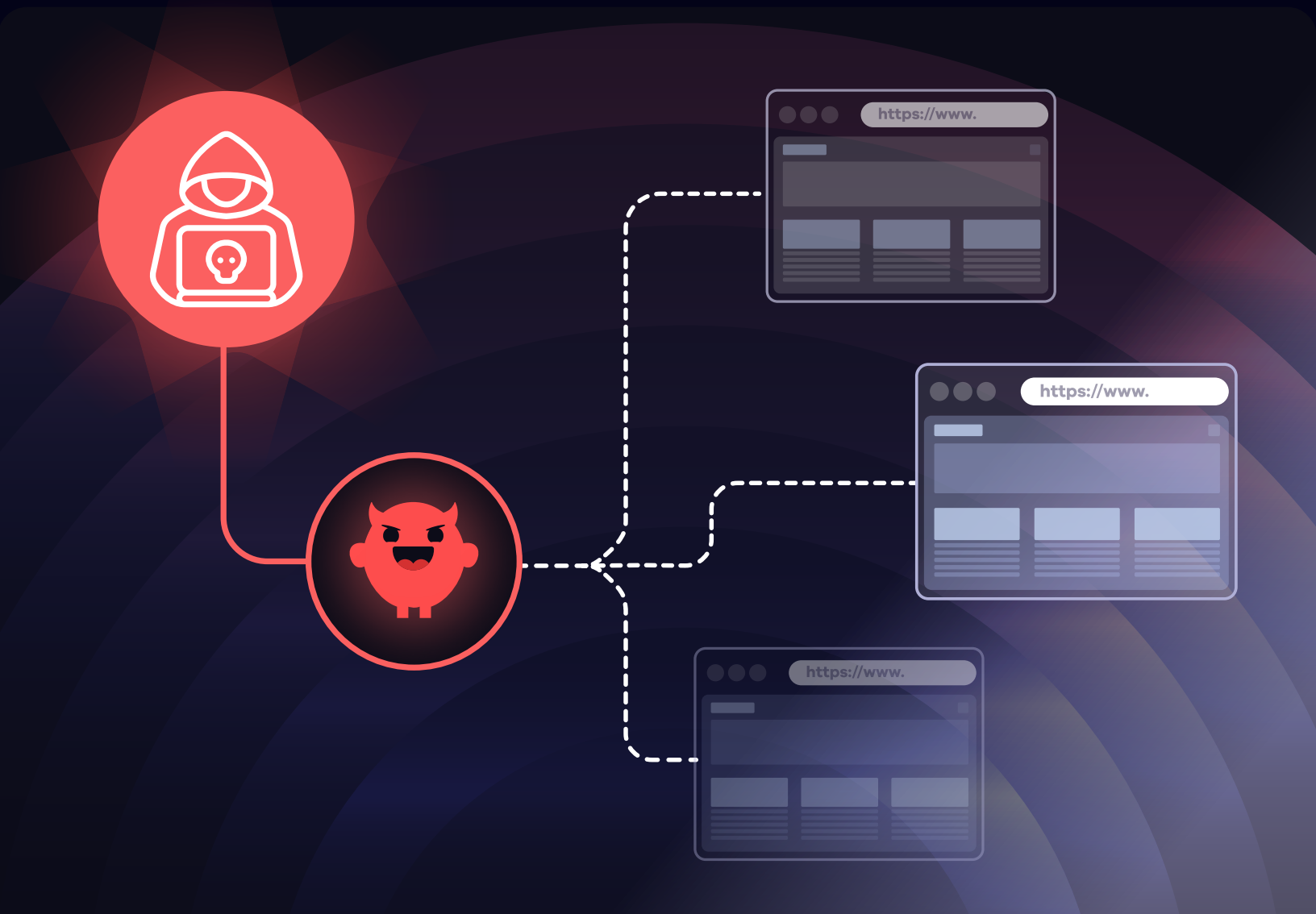
OASIS

Your Browser Is a Backdoor to Your AI Agent

A Technical Analysis of Cross-Origin WebSocket Exploitation in OpenClaw

Elad Luz
Research Lead

Published on
February 26, 2026



Overview

Introduction: What Is OpenClaw?

OpenClaw is an open-source AI personal assistant, originally created by Austrian developer Peter Steinberger under the name ClawdBot / MoltBot. Steinberger released OpenClaw in January 2026. The project's growth was unprecedented: it went from 9,000 to over 100,000 GitHub stars in just five days, making it one of the fastest-growing open-source projects in history. It currently has over 200,000 stars and an active community of thousands of developers. On February 15, 2026, Sam Altman announced that Steinberger had joined OpenAI, calling him "a genius with a lot of amazing ideas about the future of very smart agents."

OpenClaw is a self-hosted AI agent that runs on a user's machine and connects to their digital life - messaging apps (Telegram, Slack, Discord, WhatsApp), calendars, files, and development tools. Users interact with it through a web dashboard or terminal, and the agent can autonomously take actions on their behalf: send messages, run commands, search the web, manage workflows, and execute code.

The project has already faced security challenges. Within weeks of its explosive growth, researchers discovered over 1,000 malicious skills in OpenClaw's community marketplace (ClawHub) - fake plugins that deployed info-stealers and backdoors. That crisis was a supply-chain problem involving community-developed extensions.

The vulnerability described in this paper is fundamentally different. It affects the **core OpenClaw system itself** - no plugins, no extensions, no marketplace. Just the bare gateway, running exactly as documented. For many organizations, OpenClaw installations represent a growing category of shadow AI: developer-adopted tools that operate outside IT's visibility, with broad access to local systems and credentials, and no centralized governance.

Architecture: The Gateway and Nodes

OpenClaw's architecture centers on two primary components.

The **gateway** is the central coordinator. It runs as a local WebSocket server, listening by default on port **18789** on the loopback interface (127.0.0.1). The gateway handles authentication, manages chat sessions with the AI model, stores configuration (including API keys for AI providers and messaging platforms), orchestrates message routing, and exposes an RPC-style API over WebSocket for all client interactions.

Connected to the gateway are **nodes** - companion applications running on other devices. These can be the macOS desktop app, an iOS or Android device, or other machines. Nodes register with the gateway and expose device-specific capabilities: running system commands, accessing the camera, reading contacts, capturing screenshots, and more.

Clients connect to the gateway by opening a WebSocket to `ws://127.0.0.1:18789/ws`. Authentication is handled via either a **token** (a long random string) or a **password** chosen by the user. Each client identifies itself with a client ID and mode, and is granted scopes that determine which API methods it can call.

Background: CORS, WebSockets, and the Localhost Gap

The Same-Origin Policy and CORS

Web browsers enforce the **Same-Origin Policy (SOP)** - a fundamental security mechanism that prevents JavaScript on one website from reading data from a different origin. When a webpage at `https://example.com` tries to make an HTTP request to `https://another-site.com`, the browser blocks it unless the target server explicitly permits it through **Cross-Origin Resource Sharing (CORS)** headers.

Without CORS, the web would be a dangerous place. Any webpage you visit could silently make requests to your banking site, your email, or any internal service - and read the responses. CORS exists precisely to prevent this: the target server must opt in to cross-origin access by returning headers like `Access-Control-Allow-Origin`.

WebSockets: The CORS Exception

WebSocket connections are different. When a browser opens a WebSocket (`new WebSocket("ws://...")`), it starts with an HTTP upgrade request - but critically, **the browser does not enforce CORS on WebSocket connections**. There is no preflight request. There is no Access-Control-Allow-Origin check. The server receives the Origin header in the upgrade request, but it is entirely up to the server to validate it. Most WebSocket servers do not.

This means that JavaScript running on **any webpage** can open a WebSocket connection to any host and port that is reachable from the browser - including `127.0.0.1`, the user's own machine.

Browser-Specific Localhost Behavior

Browsers are gradually introducing protections for local network access, but coverage is inconsistent and WebSockets remain a significant gap:

Chrome / Edge (Chromium-based): Starting with Chrome 142 and Edge 143 (October 2025), a Local Network Access (LNA) permission prompt is shown when a public HTTPS page makes `fetch()` or subresource requests to local/loopback addresses. However, **WebSockets are explicitly not covered by LNA yet**. Chrome's documentation states that WebSocket, WebTransport, and WebRTC connections to the local network are "not yet gated on the LNA permission" and will be addressed "soon," with no specific version announced. Additionally, Chrome treats loopback addresses as secure origins, so `ws://127.0.0.1` from an HTTPS page is allowed as a mixed-content exception.

Firefox: Firefox began rolling out its own Local Network Access restrictions in Firefox 147 (January 2026), including a permission prompt for local network access. Firefox's implementation does include WebSocket checks. However, this feature is **only enabled by default for users with Enhanced Tracking Protection (ETP) set to Strict mode** and is being rolled out progressively - meaning most Firefox users do not yet benefit from this protection.

Safari: Safari is the strictest browser in this regard. It blocks insecure WebSocket connections (`ws://`) from secure HTTPS pages as mixed content, even to loopback addresses. Safari does **not** implement the loopback mixed-content exception that Chrome and Firefox do. A public HTTPS page attempting `ws://127.0.0.1` will be blocked. However, Safari has not implemented any LNA permission prompt.

Edge: Follows Chrome's Chromium behavior exactly. WebSockets to localhost are allowed, and the LNA permission prompt does not cover WebSocket connections.

The Practical Reality

As of early 2026, a webpage served over HTTPS can silently open a WebSocket to ws://127.0.0.1:18789 in Chrome, Edge, and most Firefox configurations - without any user prompt, warning, or permission dialog. The user sees nothing. Safari is the notable exception, blocking the connection as mixed content.

This creates the attack surface exploited in this paper: any website a user visits can attempt to communicate with locally running services, including the OpenClaw gateway.

The Vulnerability: Three Compounding Issues

The vulnerability chains three distinct security weaknesses, each insufficient on its own but devastating in combination.

Issue 1: Origin Validation Only Applies to Specific Client Types

When a WebSocket client connects to the OpenClaw gateway, it must identify itself with a **client ID** and **mode**. The gateway recognizes a fixed set of hardcoded client IDs, each with a specific role:

- `openclaw-control-ui` - Web dashboard
- `webchat / webchat-ui` - Embedded chat widget
- `cli` - Command-line interface
- `node-host` - Node companion process
- `openclaw-macos / openclaw-ios / openclaw-android` - Native apps
- `gateway-client` - Programmatic gateway client
- `test` - Testing

The gateway validates the `Origin` header - but **only for the `openclaw-control-ui` and `webchat` client IDs**. All other client IDs bypass origin validation entirely. This is a design choice: the CLI, native apps, and node-hosts don't originate from a web browser, so origin checking was considered unnecessary for them.

An attacker's JavaScript can simply identify as `cli` with mode `cli` - a perfectly valid combination - and the gateway will not check where the connection originated. The browser does send the `Origin` header (e.g., `https://malicious.example.com`), but the gateway ignores it for this client type.

Issue 2:
Silent Device Pairing for Localhost Connections

Separately from the origin check in Issue 1, the gateway uses a **completely different mechanism** to determine whether a connection is "local": it examines the **TCP source address** of the underlying socket. If the TCP source is a loopback address (`127.0.0.1` or `::1`), the connection is classified as a local client - regardless of what the `Origin` header says.

This is where two different security mental models collide. In web security, a connection's trustworthiness is determined by the **origin of the webpage** that initiated it - `https://malicious.example.com` is a remote, untrusted origin. But at the network layer, a browser running on the same machine connects from `127.0.0.1` - which looks identical to a legitimate local tool like the CLI or the macOS companion app. The gateway has no way to distinguish between them.

This localhost classification has a direct consequence for device pairing. Normally, when a new device connects to the gateway for the first time, a pairing request is created that r**equires explicit user approval** - the user must confirm they trust this new device. For connections classified as local, this check is bypassed entirely: pairing is marked as "silent" and auto-approved without any user interaction. The attacker's freshly generated device identity is accepted immediately, with no prompt or notification to the user.

Issue 3:
No Rate Limiting on Password Authentication for Localhost

The same TCP-based localhost classification from Issue 2 affects a second, independent security mechanism: the authentication rate limiter. The gateway implements standard brute-force protections - 10 attempts per 60-second window, with a 5-minute lockout after exceeding the limit. However, these protections **completely exempt loopback addresses** by default. Failed password attempts from `127.0.0.1` are not counted, not throttled, not recorded, and do not trigger any lockout.

With origin checks bypassed (Issue 1), device pairing auto-approved (Issue 2), and rate limiting disabled (Issue 3), an attacker can attempt password guesses at maximum speed. In lab testing, we achieved a sustained rate of **over 300 password attempts per second** from browser JavaScript alone - each attempt involving a full WebSocket connection, challenge-response handshake, Ed25519 signature, and authentication exchange.

At this rate:

- A list of 100 common passwords is exhausted in under one second
- A 10,000-entry dictionary takes approximately 30 seconds
- A 100,000-entry comprehensive wordlist takes roughly five minutes

A human-chosen password does not stand a chance against this rate of attack. The standard rate limits would be effective if applied - but they are entirely bypassed for localhost connections.

Post-Authentication: Available Methods

Once authenticated with admin-level scopes, the attacker has access to the full gateway RPC API. Below is a categorized listing of available methods and their security implications.

Agent Interaction

Method	Description
`chat.send`	Send a message to the AI agent and trigger a response - effectively hijacking the agent to process attacker-controlled prompts
`chat.history`	Read the full conversation history of any session, including user messages and agent responses
`chat.abort`	Abort an in-progress agent response

Configuration and Secrets

Method	Description
`config.get`	Retrieve the full gateway configuration, including structure of all configured services (secrets are redacted server-side but field paths remain visible)
`config.set / config.patch / config.apply`	Modify the gateway configuration—potentially redirecting traffic, changing AI providers, or altering security settings

Session Management

Method	Description
`sessions.list`	List all chat sessions with metadata
`sessions.preview`	Preview recent message snippets from sessions
`sessions.patch`	Modify session properties
`sessions.delete`	Delete chat sessions
`sessions.reset`	Reset session state

Node and Device Management

Method	Description
`node.list`	Enumerate all connected nodes with platform, IP, and device information
`node.describe`	Get detailed capabilities of a specific node
`node.invoke`	Dispatch a command to a connected node (subject to the node's security policy)
`device.pair.list`	List all paired and trusted devices
`device.pair.approve` / `device.pair.reject`	Approve or reject pending device pairing requests
`device.token.rotate` / `device.token.revoke`	Manage device authentication tokens

Agent Configuration

Method	Description
`agents.list`	List configured agents
`agents.create / agents.update / agents.delete`	Manage agent definitions
`agents.files.list/ agents.files.get / agents.files.set`	Read and write agent workspace files including identity definitions, memory, and tool configurations

Automation and Scheduling

Method	Description
`cron.list / cron.status`	View scheduled automation jobs
`cron.add / cron.update / cron.remove`	Manage scheduled jobs—an attacker could install persistent automated tasks
`cron.run`	Trigger a scheduled job immediately

Monitoring and Intelligence

Method	Description
`logs.tail`	Read recent application logs
`usage.status / usage.cost`	View usage and cost information for AI providers
`sessions.usage / sessions.usage.timeseries`	Detailed per-session usage statistics and activity timelines
`channels.status`	View connected messaging channel status

Communication

Method	Description
`send`	Send a message through a connected channel (such as Telegram, Slack, or similar services) as the user's bot
`browser.request`	Proxy HTTP requests through a connected browser node

Security Controls

Method	Description
`exec.approvals.node.get / exec.approvals.node.set`	View and modify execution approval policies on connected nodes
`skills.install / skills.update`	Install or update skills from the marketplace

System

Method	Description
`health`	Gateway health status, version, and uptime
`models.list`	Available AI models
`update.run`	Trigger a gateway update

Remediation

For OpenClaw (Implemented)

The OpenClaw security team classified this vulnerability as **High severity** and shipped a fix in version **2026.2.25** within less than 24 hours of the report, an impressive response for a volunteer-driven open-source project.

For Organizations

1. **Gain visibility into AI tooling.** You can't secure what you can't see. Inventory which AI agents and assistants are running across your developer fleet. OpenClaw instances listening on port 18789, local LLM servers, and similar tools represent a growing blind spot.

2. **If OpenClaw is installed - update immediately.** The fix for this vulnerability is included in version 2026.2.25 and later. Ensure all instances are updated. Treat this with the same urgency as any critical security patch.

3. **Review the access granted to AI agents.** OpenClaw agents can hold API keys for AI providers, connect to messaging platforms, and execute system commands on connected devices. Audit what credentials and capabilities each instance has been granted, and apply the principle of least privilege.

4. **Establish governance for non-human identities.** AI agents are a new class of identity in your organization - they authenticate, hold credentials, and take autonomous actions. They need to be governed with the same rigor as human users and service accounts.

Managing AI Agent Access at Scale

This vulnerability highlights a fundamental challenge: AI agents operate with standing privileges, long-lived credentials, and broad access,often with no centralized oversight. Traditional identity and access management wasn't designed for autonomous agents that can be hijacked through a browser tab.

This is exactly the problem that **Oasis Security's Agentic Access Management (AAM)** was built to solve. Oasis AAM provides organizations with purpose-built controls for governing AI agents across their lifecycle:

- **Intent analysis:** understanding what each agent action is trying to do before it happens, translating prompts and tool calls into auditable, structured intents
- **Policy enforcement:** applying deterministic guardrails that block dangerous actions, limit access scope, and require human approval for sensitive operations
- **Just-in-time identities:** replacing standing credentials with short-lived, per-session identities scoped only to the required task, automatically revoked when the task completes
- **Full audit trail:** maintaining a complete chain of custody from human to agent to action to result, giving security teams clear visibility and investigators a clean evidence trail

In a scenario like the one we demonstrated, Oasis AAM would enforce that even a compromised agent session operates within tightly scoped, ephemeral permissions - limiting the blast radius of any hijack and ensuring every action is logged and attributable.

As AI agents become standard tools in every developer's workflow, the question isn't whether to adopt them, it's whether you can govern them. Visibility, least privilege, and purpose-built agent access management are no longer optional.

About Oasis Security

Oasis Security is the identity security platform for the Agentic Access era.

As enterprises adopt AI at scale, they face a new security challenge: thousands of machine identities and autonomous agents operating at machine speed, without the SSO, MFA, and governance controls that protect human access.

Oasis delivers unified discovery, policy intelligence, and lifecycle enforcement across hybrid environments, giving security teams the visibility to find what legacy tools miss, the context to understand what actually matters, and the automation to govern at the speed of AI.
Backed by Accel, Cyberstarts, and Sequoia Capital, Oasis Security was founded in 2022 by Danny Brickman and Amit Zimerman.

The Oasis Research team

Our dedicated research team is committed to enhancing security in the field of identity. We take pride in our responsible and professional collaboration with vendors to address vulnerabilities and strengthen overall security.

The authors

Elad Luz, Research Lead at Oasis Security, LinkedIn

Elad Luz has over 20 years of research experience in fields such as software vulnerabilities, reverse engineering, network protocol analysis, threat detection, and ML. Prior to Oasis, he served as a CDR Research Lead at Wiz and as the Head of Research at CyberMDX. He has publicly disclosed over 20 vulnerabilities, demonstrating a strong commitment to enhancing security across multiple platforms.