SafeHaven

Raspberry Pi 5 Travel Security Appliance

Author: jstreet

Hardware: Raspberry Pi 5 + USB Wi-Fi dongle

Stack: Python · Flask · Flask-WTF · nmcli · arpspoof · tcpdump · arp-scan · nmap

Deployed: Directly on Raspberry Pi 5 via SSH — no local IDE

1. Overview

SafeHaven is a portable network security appliance built on a Raspberry Pi 5. It is designed for use in untrusted network environments — hotels, airports, conference venues — where connecting a device directly to public Wi-Fi exposes it to passive sniffing and man-in-the-middle attacks.

The Pi uses a USB Wi-Fi dongle (wlan1) to create a personal WPA2 hotspot. Client devices connect to this hotspot. The built-in Wi-Fi (wlan0) connects to the upstream network — hotel Wi-Fi, airport network — and shares the connection to hotspot clients via NAT. A Flask web interface handles all configuration from a browser, with no terminal access required.

The project has two operational modes:

- Internet routing — connect the Pi to an upstream network and share it across all hotspot clients.
- Network surveillance — intercept and inspect the traffic of a target device on the network using ARP spoofing, with live packet streaming to the browser and optional pcap export.

2. Motivation

The primary use case is IoT device monitoring. Smart TVs, IP cameras, and other connected home devices communicate with external servers in ways that are not visible to the user. SafeHaven makes it possible to position the Pi between a target device and its gateway, observe every connection it makes, and build an accurate picture of its network behaviour.

The system was tested on multiple IoT devices in a home lab environment, including Eufy IP cameras and a smart TV. It is also designed for travel: plugging a laptop or phone directly into a hotel network is a security risk, and SafeHaven puts a controlled Pi between the client and the untrusted network.

3. Architecture

3.1 Project layout

```
safehaven/
├── app.py              Flask application — all routes
├── templates/
│   ├── index.html      Landing page
│   ├── connect.html    Upstream Wi-Fi credentials form
│   ├── monitor.html    ARP spoof form + link to scan
│   ├── scan.html       arp-scan + nmap output
│   └── stream.html     Live tcpdump feed + stop/save/reset
├── static/
```

```
│   └── style.css
└── requirements.txt
```

3.2 Network interfaces

```
wlan0  Built-in Wi-Fi      Uplink — connects to hotel / home network
wlan1  USB Wi-Fi dongle    Hotspot AP — SafeHaven (10.42.0.1/24)
```

3.3 Network topology

```
[ Phone / Laptop / Tablet ]
          |
          |  Wi-Fi — SSID: SafeHaven
          ▼
  [ Raspberry Pi 5 ]
    wlan1 = hotspot AP  (10.42.0.1/24)
    wlan0 = uplink      (hotel / airport Wi-Fi)
          |
          ▼
      Internet
```

3.4 Technology stack

`Flask`	Web framework and route handling
`Flask-WTF`	Form handling and CSRF protection
`nmcli`	Hotspot creation and upstream Wi-Fi management
`arpspoof`	ARP cache poisoning (dsniff package)
`tcpdump`	Live packet capture and pcap writing
`arp-scan`	Fast ARP-based host discovery
`nmap`	Host discovery cross-check
`SSE`	Server-Sent Events — live stream to browser, no SocketIO

4. Hotspot Configuration

The hotspot is created with nmcli using ipv4.method shared on wlan1 (the USB dongle). This mode handles DHCP via a built-in dnsmasq instance, adds iptables NAT rules, and enables IP forwarding automatically — no separate hostapd config or manual sysctl entry is needed.

```
nmcli con add \
  type wifi ifname wlan1 con-name SafeHaven ssid SafeHaven \
  mode ap ipv4.method shared ipv4.addresses 10.42.0.1/24 \
  wifi-sec.key-mgmt wpa-psk wifi-sec.psk 'Drunken$ailor' \
  connection.autoconnect yes
```

With autoconnect yes, NetworkManager brings the hotspot up on every boot automatically. The Flask app starts via a systemd service after NetworkManager. A dispatcher script adds an iptables DNAT rule that redirects all HTTP traffic on wlan1 to Flask on port 80, so opening any URL in a browser on the hotspot lands on the SafeHaven interface.

5. Internet Routing

From the /connect route, the operator enters the SSID and password of the upstream network. Flask deletes any existing UpstreamWifi profile, creates a new one, and connects wlan0 to it via nmcli. NetworkManager's shared mode on wlan1 handles the NAT forwarding automatically — all hotspot clients immediately have internet access without any additional iptables configuration.

```
nmcli con delete UpstreamWifi
nmcli con add type wifi ifname wlan0 con-name UpstreamWifi \
  ssid <ssid> wifi-sec.key-mgmt wpa-psk wifi-sec.psk <password>
nmcli con up UpstreamWifi
```

Multiple devices can connect to the SafeHaven hotspot simultaneously and all share the upstream connection. New devices only need the hotspot credentials — they never interact with the upstream network directly.

6. Network Surveillance

6.1 How ARP spoofing works

ARP (Address Resolution Protocol) maps IP addresses to MAC addresses on a local network. It has no authentication — any device can send an ARP reply claiming to be any IP. ARP spoofing exploits this by continuously sending forged ARP replies to both the target device and the router, poisoning their caches so each sends traffic to the Pi's MAC address instead of each other's.

With IP forwarding enabled, the Pi passes the traffic through transparently. The target device and router remain connected, but all traffic between them flows through the Pi where it can be read.

6.2 Host discovery — /scan

Before starting an intercept, the operator navigates to /scan. This runs two discovery tools in parallel and displays the combined raw output in a pre-formatted block:

- arp-scan --interface=wlan0 --localnet — fast ARP sweep, returns IP and MAC addresses
- nmap -sn -T5 192.168.50.0/24 — ping sweep cross-check

The scan page shows the raw text output of both tools. The operator reads the IP addresses from this output and manually enters the target and router IPs into the monitor form. There is a link back to /monitor from the scan page.

6.3 Starting the intercept — /monitor

The /monitor route presents a form with two fields: Router IP and Target IP. On submit, Flask launches three background processes:

- arpspoof -i wlan0 -t <target> <router> — poisons the target's ARP cache
- arpspoof -i wlan0 -t <router> <target> — poisons the router's ARP cache
- tcpdump -i wlan0 -w /tmp/capture.pcap host <target> — silent pcap writer

Process handles are kept in module-level globals. After starting, the user is redirected to /stream.

6.4 Live traffic stream — /stream and /stream_feed

The /stream route renders stream.html, which opens an EventSource connection to /stream_feed passing the target IP as a query parameter. The /stream_feed route spawns a separate tcpdump process in live mode and reads its stdout line by line, yielding each line as an SSE event:

```
tcpdump -i wlan0 -l --immediate-mode host <target_ip>
```

The -l flag enables line buffering and --immediate-mode flushes output immediately, ensuring packets appear in the browser in real time. The stream.html page appends each line to a scrolling black terminal-style div. No SocketIO or WebSocket is used — SSE over plain HTTP is sufficient.

Note: this means two tcpdump processes run simultaneously — one writing the pcap silently, one streaming live output to the browser.

6.5 Stop, save, and reset

Three actions are available from stream.html:

- Stop (/stop) — sends SIGTERM to all three processes (both arpspoof instances and the pcap tcpdump). The live SSE tcpdump is closed client-side by calling es.close() before the form submits.
- Save (/save) — if /tmp/capture.pcap exists, sends it as a file download with the filename capture.pcap. The file can be opened in Wireshark for offline analysis. Saving is always optional — the pcap is written continuously while the intercept runs but is never saved unless the operator explicitly requests it.
- Reset (/reset) — terminates all processes, clears all global state, and redirects to the home page.

7. Route Summary

`GET /`	Landing page
`GET/POST /connect`	Upstream Wi-Fi credentials form
`GET/POST /monitor`	ARP spoof form
`GET /scan`	arp-scan + nmap host discovery
`GET /stream`	Live traffic page
`GET /stream_feed`	SSE feed — tcpdump stdout
`POST /stop`	Terminate arpspoof + pcap tcpdump
`POST /save`	Download capture.pcap
`POST /reset`	Kill all processes, return home

8. Deployment

8.1 Development approach

This project was developed entirely on the Raspberry Pi 5 itself via SSH — no local IDE, no PyCharm, no staging environment. Because the hotspot and network interfaces are the core of the application, it cannot be meaningfully developed or tested on any machine other than the hardware it runs on.

This is a deliberate departure from the usual workflow of developing locally in an IDE and deploying to a server. For embedded network projects, the environment is the application.

8.2 Boot sequence

On power-on the sequence is fully automatic:

- NetworkManager starts and brings up the SafeHaven hotspot on wlan1 (autoconnect yes)

- The hotspot assigns 10.42.0.1/24 to wlan1 and starts dnsmasq for DHCP
- A NetworkManager dispatcher script adds the HTTP redirect iptables rule on wlan1
- systemd starts the Flask app on port 80

A device connecting to the hotspot immediately reaches the web UI. No interaction with the Pi is required after power-on.

8.3 systemd service

```
[Unit]
Description=SafeHaven
After=network.target NetworkManager.service

[Service]
ExecStart=/opt/safehaven/venv/bin/python /opt/safehaven/app.py
Restart=on-failure
User=root

[Install]
WantedBy=multi-user.target
```

9. Testing & Results

The system was tested against multiple IoT devices on a home network:

- Eufy IP cameras — confirmed to contact only AWS European regions (Frankfurt, Stockholm, Paris) via UDP heartbeats on ports 8006 and 32100, using direct P2P WebRTC for video streams.
- Smart TV — successfully intercepted during live football streaming. The ARP poison introduced enough latency through the Pi that the video stream froze, confirming the intercept was working. High-bandwidth active streaming is not a suitable target for live MITM monitoring; idle IoT traffic is the intended use case.
- arp-scan reliably discovered all active devices on the subnet within 2–3 seconds.
- ARP cache restoration after stopping worked correctly — target devices recovered connectivity within seconds.

10. Planned Extensions

- WireGuard VPN tunnel — route all hotspot traffic through an encrypted tunnel to a home Raspberry Pi acting as a WireGuard server. The hotel network would see only encrypted UDP on port 51820. The travel Pi would be the WireGuard client (AllowedIPs = 0.0.0.0/0), the home Pi the server with NAT masquerade to the internet. A toggle in the web UI would bring the tunnel up and down via wg-quick.
- Connection blocking — iptables DROP rules per IP or domain, manageable from the UI. Natural progression from passive surveillance to active control.
- DNS-level blocking — run dnsmasq with a blocklist on the Pi for all hotspot clients, Pi-hole style.
- Scan UI improvement — replace the raw text output on /scan with a parsed table so IPs are easier to read and select.

This project is for personal use and authorised network testing only. ARP spoofing on networks you do not own or have explicit permission to test is illegal.