Independent Security Evaluation of OpenNHP (NHP)

IETF 127 Hackathon — independent reviewer submission Version tested: OpenNHP v1.0.2. Evaluation performed November 2026.

Rahul Ramachandran | rahultn@duck.com

Scope and objective

I serve as an invited independent reviewer for the OpenNHP hackathon, with a brief to assess the AI-driven attack methodology, the test coverage, the results, and the limitations and opportunities for improvement. This document is my evaluation of the protocol and the testing conducted against it to date.

I approached it the way I would any independent review: decide what the protocol claims to do, then put the most important of those claims to the test myself in a controlled lab. The aim is an honest picture of what holds up and what needs more work, with every claim backed by evidence I can point to.

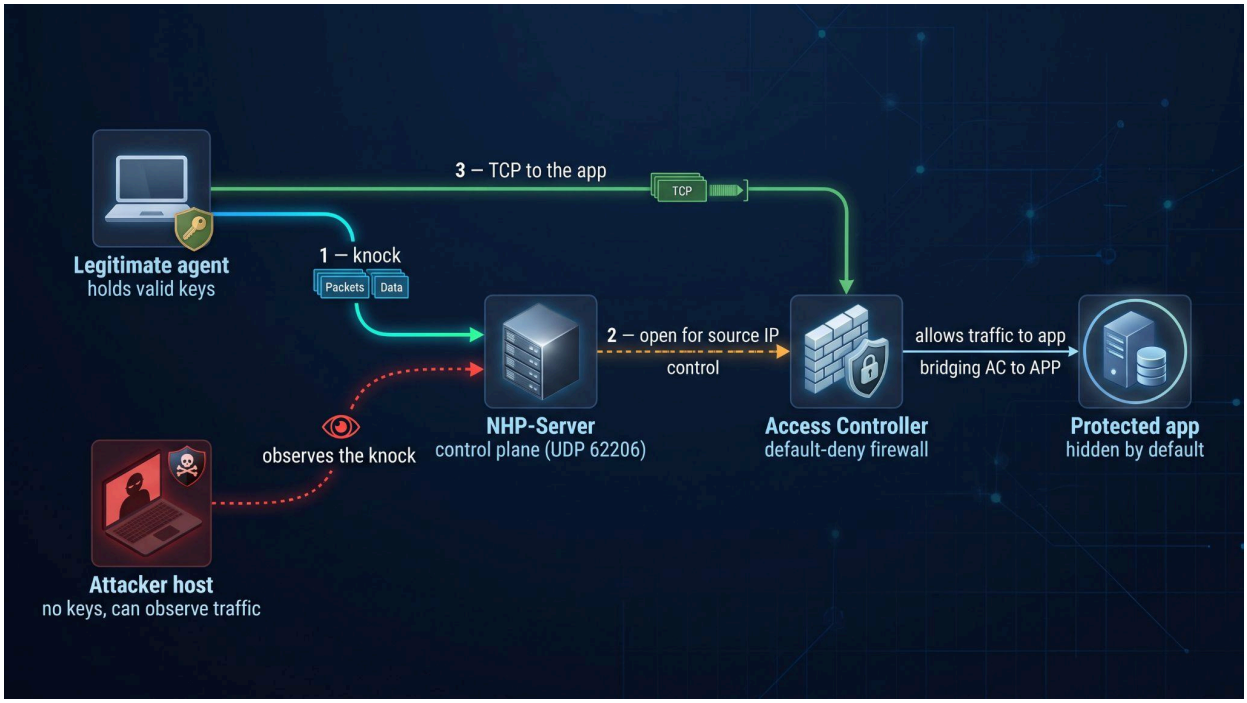
A companion document covers enterprise deployment readiness; configuration, key distribution, and what it would take for NHP to stand in for a ZTNA product. This document stays on the security evaluation.

I carried out this evaluation independently, in a personal capacity, as an invited reviewer for the IETF 127 Hackathon. The analysis and findings are my own. They do not represent the views of my employer, Palo Alto Networks, and this work is not affiliated with, endorsed by, sponsored by, or produced on behalf of Palo Alto Networks. Palo Alto Networks is not a party to this evaluation, has no involvement in or responsibility for its contents, and should not be cited, quoted, or referenced in connection with it. I performed the work on my own time and resources, against publicly available OpenNHP software, and provide it in good faith and as-is.

Method and environment

I built a self-contained private lab and ran the full OpenNHP stack from the v1.0.2 release: the control-plane server, the access controller, a protected web application behind it, and a legitimate agent. All of it sat on one shared network segment, with each role on its own address, so I could watch every packet and every firewall change directly.

The roles and the three paths between them look like this:



One honest caveat on the environment. Because every role ran on a single shared segment, I proved the behavior on a local network rather than across the public internet. The finding below turns entirely on how the server decides which address to open the door for, and that logic is the same whether the packet crossed a room or a continent. So the mechanism generalizes, though I did not measure it over the open internet.

Coverage assessment

I focused this evaluation on the protocol's core promise: that an unauthorized party cannot reach a protected service without a valid, authenticated knock. In a controlled lab running OpenNHP v1.0.2, I confirmed the default-deny behavior, then looked closely at what the access controller opens once a knock succeeds. That is where the replay finding below comes from, reproduced end to end. I also read the v1.0.2 source to understand how knocks are validated and how access is granted and withdrawn.

A few areas sit outside what I tested and would each reward a dedicated look: pre-authentication exploitation of a protected service, denial of service against the knock port, man-in-the-middle on the knock exchange, and a formal review of the custom handshake. Where I touch on these below, the observations come from reading the source rather than from testing.

What held up

The core promise works. Before any knock, the protected application was genuinely invisible from the agent's own host: a direct request timed out, and the access controller's rule set was empty. The moment a legitimate, cryptographically valid knock arrived, the server authorized it and the controller opened a short-lived path. Stop the agent, and the door closes again on its timer. Default-deny with authenticate-before-connect behaves exactly as advertised against an unauthenticated party who has no captured knock to work with.

The freshness check also does its job. When I replayed a captured knock that was about ten minutes old, the server rejected it as stale and dropped it. That is the intended backstop against old captured traffic, and it functioned.

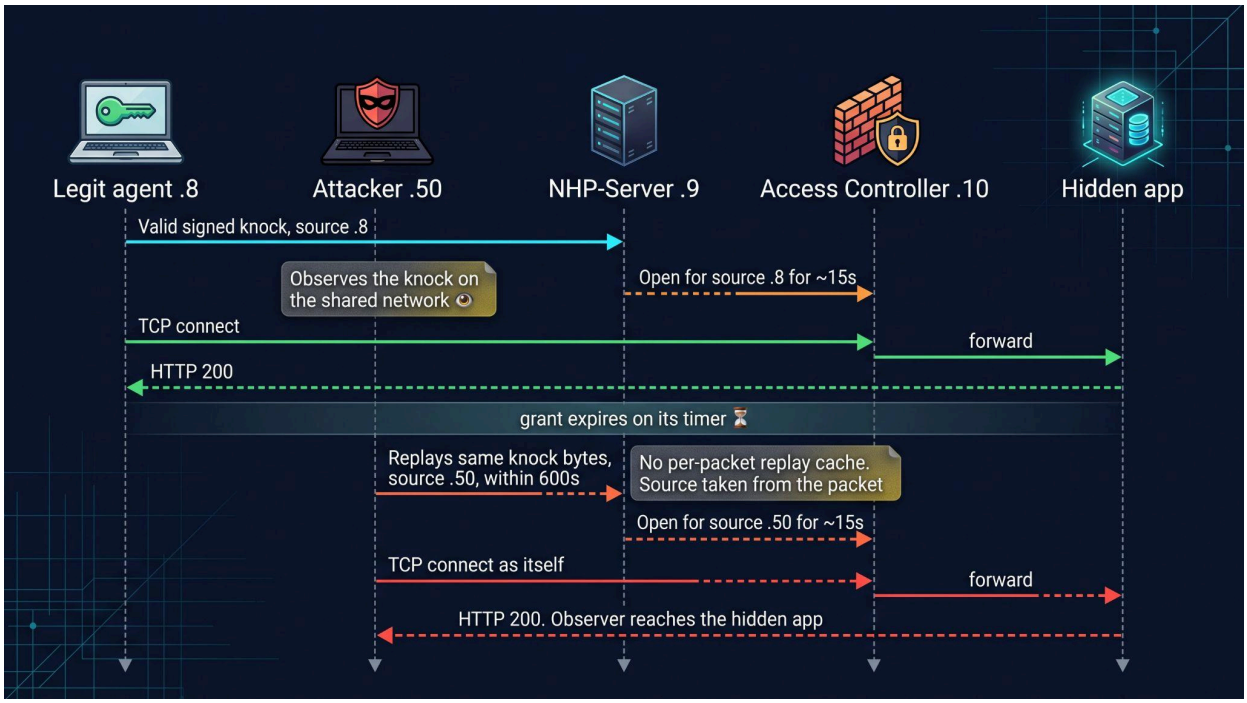
Both of these are real strengths, and they are the reason the finding in the next section is narrow and specific.

What did not hold up: knock replay binds access to the replayer

NHP authorizes a knock on the sender's cryptographic identity, which is the right idea. The gap is in what happens next. When the server approves a knock, it reads the source address straight from the arriving packet and tells the access controller to open for that address. The grant that lands in the firewall is an address, a port, and a destination, with a short timeout. It carries no memory of who actually knocked.

That has a consequence I was able to reproduce end to end. A valid knock, captured off the shared network and replayed byte-for-byte from a different host within the freshness window, opens the door for the replaying host.

Here is the sequence I observed:



A detail makes this sharper than a typical spoofing attempt. The replaying host has none of the agent's keys and cannot read the server's encrypted reply. It does not need to. Once the firewall opens for its address, it reaches the application over ordinary TCP as itself, and gets a normal, fully working connection. The cryptography that protects the knock stops mattering the instant the pinhole opens, because the data plane checks only the source address.

I verified the mechanism in the server's own logs: when the replay arrived from the new address, the server decrypted it cleanly and sent the controller an open instruction naming that new address. I then confirmed two supporting facts in the v1.0.2 source. The server keeps no replay cache for knocks; the per-packet nonce and counter in the header are carried but never checked against a record of what has already been seen, so the server reprocessed a packet it had already handled from the real agent. And the only freshness defense is a fixed 10-minute window, written directly into the code with no configuration setting to tighten it. The specification recommends 60 seconds for replay protection, so the shipped implementation runs 10x looser than its own spec advises, and an adopter cannot change that without editing the source.

What this means, and what it does not: What is defeated is NHP's central promise, that an unauthorized party cannot even reach the protected service. An attacker who can observe a knock, or who shares a source address with a legitimate agent, can reach a service that was supposed to be invisible to them. What is not defeated is the application's own TLS and login, which NHP never claimed to replace. The honest framing is the same one you would use for an SSL VPN: this gates reachability, and once the tunnel opens, the application's own protection is all that remains. That makes the exposure most serious exactly where NHP adds the most

value — in front of plaintext internal tools, legacy admin panels, or unencrypted services, which are the very things people hide behind this kind of layer.

How reachable the capture is: The attacker needs to see the knock, and seeing it is easier than a full man-in-the-middle. Passive observation is enough: a shared Wi-Fi network, a mirrored switch port, a compromised router, ARP poisoning on the same segment, a tap along the route, or even a packet log or cloud flow capture where the raw knock bytes are recorded. The shared-NAT case needs no capture at all, since a second host behind the same egress address already presents the same source. The 10-minute window bounds all of this: a captured knock is useful for that long, and within it an automated attacker can replay freely.

The spec's own replay-prevention section calls for a unique nonce per message and a configurable timestamp tolerance, recommended at 60 seconds. The v1.0.2 implementation carries the nonce without checking it, and hardcodes the tolerance at 10 minutes.

Recommendations

Each of these pairs with the finding above, and each is a concrete step the project can take;

1. **Add per-packet replay protection for knocks.** Keep a short-lived record of the nonces or packet identifiers the server has already accepted, and reject a repeat. The header already carries a nonce and a counter, so the raw material is there; it simply is not checked today.

2. **Tie the opened grant to the authenticated identity rather than the packet's source address.** The cleanest version carries the proven identity into the data-plane decision, so that a captured knock replayed from elsewhere opens nothing. This is also the natural bridge to per-request, identity-aware enforcement at the controller, which would close the shared-NAT case as well.

3. **Make the freshness window configurable and align the default with the spec.** 60 seconds matches the specification's own recommendation. Shortening the window narrows the exposure, though on its own it does not remove it, which is why it belongs alongside recommendation 1 rather than instead of it.