LINUX SECURITY PROJECT
NetGuard Zero-Trust Server
Aegis Linux

BY IDEO-LAB

MARCH 2026

IDEOLAB

CRYPTO SÉCURITÉ

2.1 Immutabilité

Le système ne doit plus être liberanre modifiable en production.

2.3 Éphemère

Secrets, accès admin, certificats, sessions ; duaée de vie minimale.

2.4 Cloisonnement

Chaque brique ne voit que le strict minimum.

2.5 Runtime enforcement

La sécurité ne s'arrête pas au boot ; elle continue pendant l'exécution.

2.6 Détection + réaction

Détecter ne suffit pas ; il faus pouvoir agir aųtomatiquemens.

2.2 Attestation

Le servere doit prouver sen état avant d'obtent confiance.

2.3 Éphemère

Secrets, accès admin, certificats, sessoms : durée de vie minimale.

2.4 Cloisonnement

Chaque brique ne voit que le strict minimum.

2.5 Reconstruction

Un servere douteux ne doit pas être 'réparé à la main", mais rempac.

2.8 Traçabilité

Tout événement important

Linux Zero-Trust Ephemeral Server

Un serveur qui n'est plus un "animal de compagnie" qu'on administre à la main, mais un **objet jetable, mesuré, verrouillé, attesté, et auto-régénérant.**

AGENDA

- 1. Système racine immuable
- 2. Boot mesuré + attestation distante
- 3. Secrets jamais stockés localement
- 4. Micro-segmentation native
- 5. Exécution cloisonnée par défaut
- 6. Admin sans SSH permanent
- 7. Détection comportementale locale
- 8. Reconstruction automatique au lieu de réparation
- 9. Données actives chiffrées et fragmentées
- 10. Noyau défensif “paranoïaque”

1. Système racine immuable

Le système de base est **read-only**, signé, vérifié au boot.
Aucune modif durable possible sur l'OS en prod.

Idée :

- rootfs immuable
- packages figés
- pas d'install "à chaud"
- rollback atomique
- toute dérive = rejet ou reconstruction

2. Boot mesuré + attestation distante

Le serveur prouve cryptographiquement au contrôleur central :

- quel kernel il exécute
- quelle image système
- quels modules
- quelle config de sécurité

Si la mesure ne correspond pas au profil attendu :

- pas de secrets
- pas d'accès réseau sensible
- quarantaine immédiate

3. Secrets jamais stockés localement

Le serveur ne possède pas ses secrets durablement.
Il les **loue** pour une durée courte après attestation.

Exemples de logique :

- certificats éphémères
- clés de session à durée de vie très courte
- rotation automatique
- révocation immédiate si comportement anormal

4. Micro-segmentation native

Chaque service voit uniquement :

- ses dépendances strictes
- son port autorisé
- ses identités approuvées

Même si un service tombe, le mouvement latéral devient très difficile.

5. Exécution cloisonnée par défaut

Au lieu de faire confiance au process :

- sandbox stricte
- seccomp/AppArmor/SELinux renforcé
- capacités Linux minimales
- namespaces isolés
- accès fichiers très limité

L'idée est qu'un process compromis ne puisse presque rien faire.

6. Admin sans SSH permanent

Le gros changement "inventif" serait de supprimer presque totalement l'admin classique.

Pas de :

- SSH ouvert en permanence
- mots de passe
- comptes admin longs termes

À la place :

- accès d'administration temporaire
- session approuvée à la demande
- double validation
- fenêtre de temps courte
- enregistrement intégral

7. Détection comportementale locale

Pas juste des signatures.

Le serveur construit un profil normal :

- appels système habituels
- arbre de processus attendu
- volumes réseau normaux
- fichiers normalement touchés

Dès qu'un écart fort apparaît :

- blocage automatique
- gel du conteneur/process
- capture forensique
- remplacement du nœud

8. Reconstruction automatique au lieu de réparation

Le futur, ce n’est pas “nettoyer” un serveur compromis.

C’est :

- détecter
- couper
- remplacer par une image saine
- réinjecter seulement l’état métier validé

Donc :

cattle, not pets, mais poussé à l’extrême sécurité.

9. Données actives chiffrées et fragmentées

Même après compromission partielle :

- données chiffrées au repos
- clés séparées du serveur
- segmentation des jeux de données
- déchiffrement juste-in-time
- accès minimal selon identité et contexte

10. Noyau défensif “paranoïaque”

Une techno à inventer pourrait être un **hyper-agent kernel/user-space** qui fait ceci :

- refuse les exécutions non approuvées
- vérifie la provenance des binaires/scripts
- bloque toute élévation non prévue
- journalise de façon inviolable
- coupe le réseau dès dérive critique

En gros, une sorte de **guardian layer** entre Linux et le monde réel.

Ce que je construirais concrètement comme techno nouvelle

PLAN GENERAL

1. Trusted Boot Controller
2. Ephemeral Secrets Broker
3. Runtime Containment Engine
4. Autonomous Rebuilder
5. Tamper-Proof Audit Ledger

Les principes d'or

Pour s'approcher du "quasi impossible à hacker" :

- moins de code
- moins de services
- moins d'humains avec accès
- moins de secrets stockés
- plus d'attestation
- plus d'éphémère
- plus d'automatisation
- plus de reconstruction
- plus de cloisonnement

La formule la plus honnête

Le but n’est pas :

“empêcher toute intrusion pour toujours”

mais :

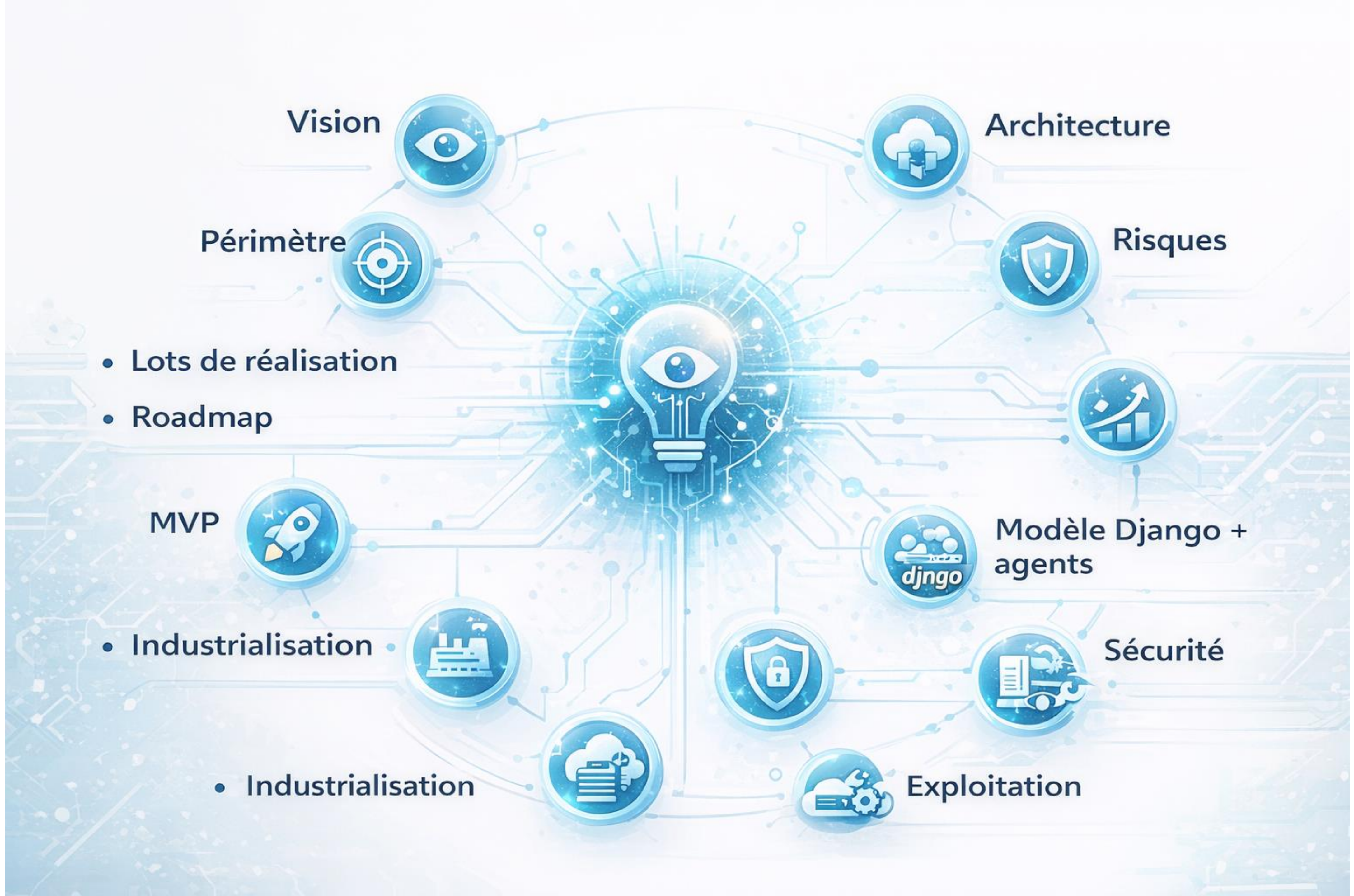
“rendre l’intrusion rare, difficile, bruyante, peu rentable, peu profonde, et rapidement réversible.”

- data model
- agent Python/Rust
- dashboard
- policies
- orchestration de quarantaine/rebuild.

Aegis Linux - NetGuard Aegis

- Trusted, Ephemeral, Self-Healing Linux Security Platform
- Zero-Trust Linux Hardening & Autonomous Recovery Platform

- vision
- périmètre
- architecture
- lots de réalisation
- roadmap
- risques
- MVP
- industrialisation
- modèle Django + agents
- sécurité
- exploitation


Vision
Architecture
Périmètre
Risques
• Lots de réalisation
• Roadmap
MVP
Modèle Django + agents
django
• Industrialisation
Sécurité
• Industrialisation
Exploitation

1. Vision du projet

Objectif

Construire une plateforme capable de transformer un serveur Linux classique en nœud :

- attesté
- durci
- très cloisonné
- surveillé en continu
- à secrets éphémères
- auto-isolable
- reconstructible automatiquement

But réaliste

Pas “impossible à hacker”.

Mais :

- extrêmement difficile à compromettre
- compromission rapidement détectée
- secrets non persistants
- blast radius minimal
- reconstruction rapide
- audit sérieux

Cible

- serveurs Linux Internet exposés
- reverse proxy / Nginx
- API backends
- workers
- bastions d’administration
- petits clusters
- éventuellement VM, bare metal, cloud

2.1 Immutabilité

Le système ne doit plus être librement modifiable en production.

2.2 Attestation

Le serveur doit prouver son état avant d’obtenir confiance.

2.3 Éphémère

Secrets, accès admin, certificats, sessions : durée de vie minimale.

2.4 Cloisonnement

Chaque brique ne voit que le strict minimum.

2.5 Runtime enforcement

La sécurité ne s’arrête pas au boot ; elle continue pendant l’exécution.

2.6 Détection + réaction

Détecter ne suffit pas ; il faut pouvoir agir automatiquement.

2.7 Reconstruction

Un serveur douteux ne doit pas être “réparé à la main”, mais remplacé.

2.8 Traçabilité

Tout événement important doit être exporté hors du nœud et conservé.

Principes de sécurité essentiels

2.1 Immutabilité

Le système ne doit plusèe être librement modifiable en production.

2.2 Attestation

Le serveur doit prouver son état avant d'obtenir confiance.

2.3 Éphémère

Secrets, accès admin, certificats, sessions : durée de vie minimale.

2.4 Cloisonnement

Chaque brique ne voit que le strict minimum.

2.5 Runtime enforcement

La sécurité ne s'arrête pas au boot ; elle continue pendant l'exécution.

2.6 Détection + réaction

Détecter ne suffit pas ; il faut pouvoir agir automatiquement.

2.7 Reconstruction

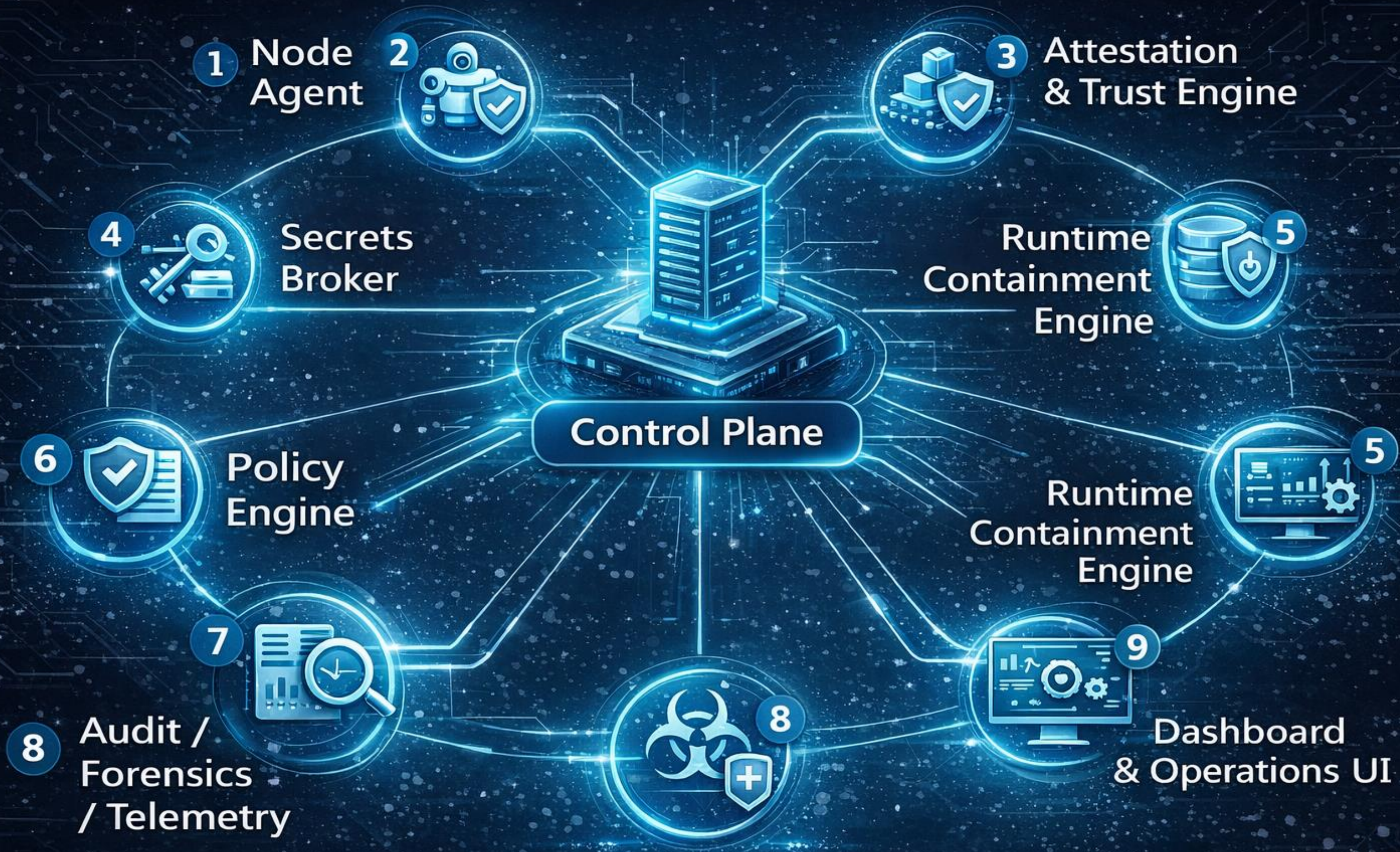
Un serveur douteux ne doit pas être "réparé à la main", mais remplacé.

2.8 Traçabilité

Tout événement important doit être exporté hors du nœud et conservé.

3. Grand découpage du programme

1. Control Plane
2. Node Agent
3. Attestation & Trust Engine
4. Secrets Broker
5. Runtime Containment Engine
6. Policy Engine
7. Quarantine & Recovery Engine
8. Audit / Forensics / Telemetry
9. Dashboard & Operations UI



1
Node
Agent
2
3
Attestation
& Trust Engine
4
Secrets
Broker
Runtime
Containment
Engine
5
Control Plane
6
Policy
Engine
5
Runtime
Containment
Engine
7
8
Audit /
Forensics
/ Telemetry
8
9
Dashboard
& Operations UI

4. Architecture cible

Côté serveur protégé

Chaque nœud Linux embarque un **agent Aegis** qui :

- collecte son état
- applique des politiques
- surveille les processus
- gère l'isolement
- remonte les événements
- reçoit des ordres du control plane

Côté central

Un **control plane Django** :

- stocke inventaire, état, politiques, incidents
- prend les décisions
- expose une API sécurisée
- pousse des ordres aux agents
- gère secrets éphémères, attestations, quarantaines, rebuilds

Côté exécution infra

Une couche d'orchestration :

- scripts système
- Ansible / Terraform / cloud-init
- éventuellement Rust/Python workers
- outils de reprovisionnement

Côté serveur protégé

Chaque nœud Linux embarque un **agent Aegis** qui :

- collecte son état
- applique des politiques
- surveille les processus
- gère l'isolement
- remonte les événements
- reçoit des ordres du control plane

Côté central

Un control plane Django :

- stocke inventaire, état, politiques, incidents
- prend les décisions
- expose une API sécurisée
- pousse des ordres aux agents
- gère secrets éphémères, attestations, quarantaines, rebuilds

Côté central

Côté serveur protégé

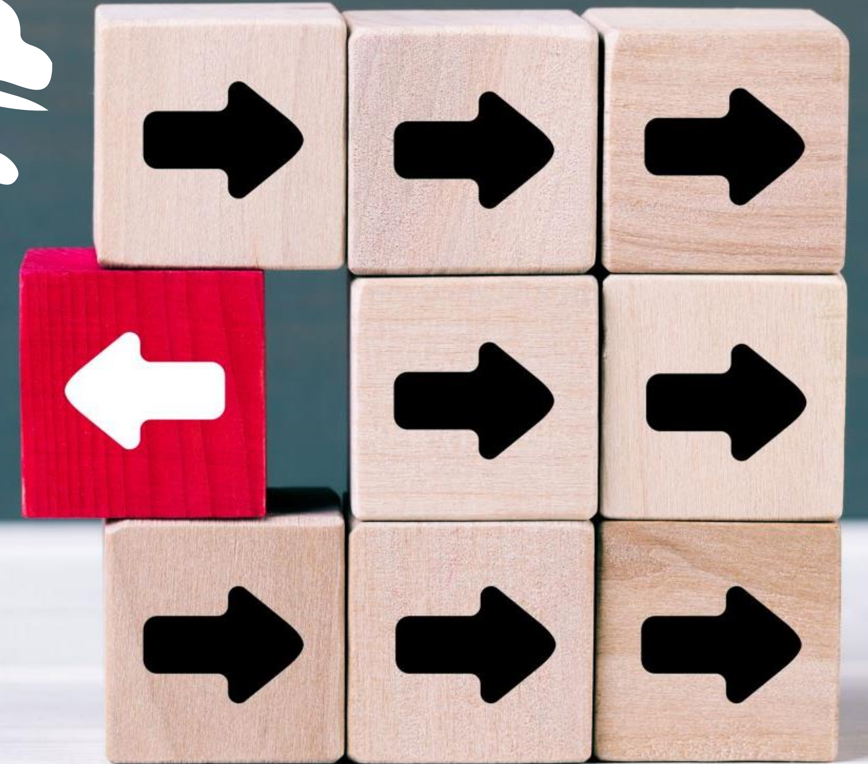
- G'cele son état
- applique des politiques
- surveille les processus
- gère l'isolement
- remonte les événements
- reçoit des ordres du control plane

Côté exécution infra

Une couche d'orchestration :

- scripts système
- Ansible / Terraform / cloud-init
- éventuellement Rust/Python workers
- outils de reprovisionnement

5. CHOIX TECHNOLOGIQUES



5. Choix technologiques recommandés

5.1 Backend central

5.2 Agent nœud

5.3 Mécanismes Linux

5.1 Backend central

- **Django** pour le control plane
- **PostgreSQL** pour la base centrale
- **Redis** pour files/caching/événements légers
- **Celery** pour tâches asynchrones
- DRF pour API
- **Nginx** frontal
- **WebSocket / SSE** pour dashboards live si besoin

Django
Control Plane
Contróle l'ensemble des processus

PostgreSQL
Base de données centrale
Stockage des données et metadata

Redis
Files, cache & évenements légers
Segments cache, files et evénements

Celery
Tâches asynchrones
Exécution en arriére-plan des jobs

DRF
API REST sécurisé
Interface sécurisée pour les agents

Nginx
Serveur frontal
Gestion du trafic et reverse proxy

WebSocket / SSE
Dashboards live
Mise à jour en temps réel des UI

5.2 Agent nœud

Option A — démarrage rapide

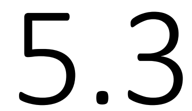
- agent en **Python**
- plus rapide à produire
- meilleur MVP
- meilleure vitesse d'itération

Option B — industrialisation

- parties critiques en **Rust**
- agent hybride Python + Rust
- Python pour orchestration
- Rust pour surveillance, perf, parsing logs, modules sensibles

Ma reco :

- **Phase 1 à 3 : Python**
- **Phase 4+ : migration partielle vers Rust**



5.3 Mécanismes Linux

- systemd
- nftables
- journald
- auditd
- eBPF si on veut aller loin
- seccomp
- AppArmor ou SELinux
- fs-verity / dm-verity selon ambition
- TPM si disponible
- SSH fortement réduit ou remplacé par accès éphémère

5.3 Mécanismes Linux

- systemd
- nftables
- journald
- auditd
- eBPF si on veut aller loin
- seccomp
- AppArmor ou SELinux
- fs-verity / dm-verity selon ambition
- TPM si disponible
- SSH fortement réduit ou remplacé par accès éphémère

6. Lots du projet

AGENDA DU PROJET

- LOT 0 — Cadrage et doctrine
- LOT 1 — Socle Django du Control Plane
- LOT 2 — Agent Linux V1
- LOT 3 — Enrôlement et identité machine
- LOT 4 — Attestation et chaîne de confiance
- LOT 5 — Policy Engine
- LOT 6 — Runtime Containment
- LOT 7 — Secrets Broker
- LOT 8 — Quarantaine et réaction
- LOT 9 — Recovery / Rebuild
- LOT 10 — Audit, forensic, compliance
- LOT 11 — Dashboard applicatif

Phase 1 — Foundations

But

Poser le socle.

Contenu

- projet Django
- modèles initiaux
- API node enrollment
- heartbeat
- inventaire
- dashboard simple
- agent Python minimal

Sortie attendue

On voit les nœuds, leur état, leur inventaire.

Phase 2 — Hardening & Policies

But

Commencer à agir sur les nœuds.

Contenu

- policy engine V1
- application de hardening
- règles SSH
- sysctl
- services autorisés
- nftables baseline
- logs sécurité

Sortie attendue

Les nœuds convergent vers un état connu.

Phase 3 — Detection & Incidents

But

Détecter les dérives.

Contenu

- process watch
- auth watch
- conf integrity watch
- incidents
- scoring
- dashboard incidents
- soft quarantine

Sortie attendue

On sait détecter et classifier.

Phase 4 — Secrets & Trust

But

Réduire radicalement le risque lié aux secrets.

Contenu

- broker secrets éphémères
- mTLS agents
- trust gating
- blocage secrets si attestation faible

Sortie attendue

Un nœud douteux ne reçoit plus de valeur sensit

Phase 5 — Automated Containment

But

Réagir automatiquement.

Contenu

- medium/hard quarantine
- nftables auto
- blocage egress
- kill process
- isolation playbooks

Sortie attendue

Le système sait contenir.

Phase 6 — Rebuild & Self-Healing

But

Passer de la défense à la résilience.

Contenu

- workflows rebuild
- intégration cloud-init / terraform / ansible
- destruction + reprovision
- réintégration contrôlée

Sortie attendue

Un nœud compromis devient remplaçable.

Phase 7 — Advanced Trust

But

Industrialiser.

Contenu

- TPM
- measured boot
- eBPF avancé
- forensic amélioré
- policy tuning
- Rust modules critiques

Sortie attendue

Plateforme mature, très haut niveau.

8. MVP recommandé

Pour éviter un projet énorme dès le début, voici le **MVP réaliste**.

MVP = 6 capacités seulement

1. enrollment nœud
2. heartbeat + inventaire
3. policy hardening simple
4. remontée événements sécurité
5. soft quarantine
6. dashboard + incidents

Ce qu'on ne fait pas tout de suite

- TPM complet
- eBPF avancé
- HSM
- rebuild bare metal complexe
- IA comportementale lourde
- kernel custom

Decoupage du projet

AGENDA
DECOUPAGE

9.1 Apps Django détaillées

9.2 Agent Linux modules détaillés

9. Découpage technique plus fin

`aegis_runtime`

État runtime, anomalies, confinement.

`aegis_secrets`

Distribution courte durée.

`aegis_recovery`

Quarantaine, rebuild, historique.

`aegis_audit`

Piste d'audit inviolable logique.

`aegis_dashboard`

UI staff / ops.

`aegis_nodes`

Inventaire, heartbeat, identité.

`aegis_policy`

Définition et assignation des politiques.

`aegis_attestation`

Rapports, score de confiance, dérives.

`aegis_events`

Événements entrants, normalisation.

`aegis_incidents`

Moteur d'incident, workflow SOC léger.

9.2 Agent Linux modules détaillés

Identity

- bootstrap
- certs
- rotation

Inventory

- os
- kernel
- packages
- services
- network listeners

Integrity

- hashes fichiers
- systemd units
- conf security

Runtime

- processes
- sessions
- ports
- sudo/auth

Policy

- fetch
- validate
- apply
- report

Transport

- mTLS
- retry
- spool local

Quarantine

- nftables
- stop services
- lock admin

Forensics

- collect artefacts
- compress
- upload

10. Sécurité du control plane lui-même

Très important :

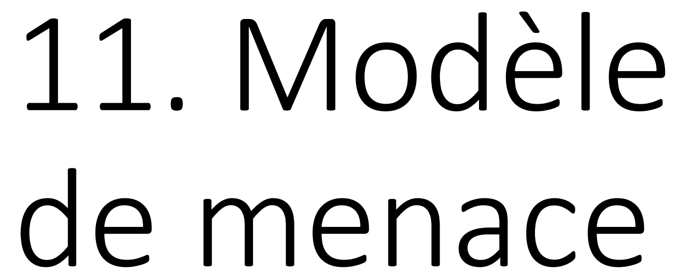
le control plane devient une cible majeure.

Obligations

- segmentation dédiée
- accès staff très réduit
- MFA forte
- journaux complets
- chiffrement au repos
- sauvegardes
- rotation clés
- séparation des rôles
- bastion admin dédié
- review des actions critiques

Ne jamais faire

- laisser le control plane géré comme un banal backoffice Django



Menaces visées

- brute force
- credentials volés
- malware utilisateur
- webshell
- persistence système
- dérive config
- service exposé non autorisé
- exfiltration
- latéralisation
- administrateur compromis
- supply chain logicielle partielle

Menaces peu ou pas couvertes au début

- firmware matériel avancé
- adversaire nation-state avec accès physique long
- microcode compromis
- chaîne de boot intégralement subvertie sans TPM/secure boot

Menaces visées

Brute force

Le système ne doit plus être librement modifiable en production.

Credentials volés

Le serveur doit prouver son état avant d'obtenir

Webshell

Persistance systéme

Dérive config

Latéralisation

Admintrateur compromis

Persistence systéme

La sours rcontinne

Dérive config

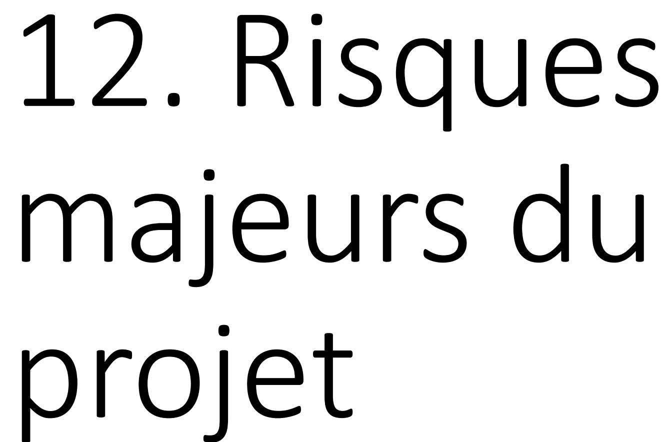
LaCCS qui persistent

Service exposé non autorisé

Exfiltration

Administrateur compromis

Supply chain logicielle partielle



12. Risques majeurs du projet

12.1 Faux positifs

Un moteur trop agressif casse la prod.

12.2 Complexité opérationnelle

Un système ultra-sécurisé mais incompréhensible finit contourné.

12.3 Centralisation du risque

Le control plane devient critique.

12.4 Dette policy

Trop de règles mal gérées = chaos.

12.5 Rebuild dangereux

Un rebuild mal pensé peut effacer trop ou réintroduire un état sale.

12.6 Dépendance agent

Si l'agent est faible ou contournable, la vision centrale devient trompeuse.

Risques Identifiés

12.1 Faux positifs

Un moteur trop agressif
casse la prod.

12.2 Complexité opérationnelle

Un système ultra-sécurisé mais
incompréhensible finit contourné.

12.3 Centralisation du risque

Le control plane devient critique.

12.4 Dette policy

Trop de règles mal gérées = chaos.

12.5 Dette policy

Trop de règles mal pensé
= chaos.

12.5 Rebuild dangereux

Un rebuild mal pensé peut effacer
trop ou réintroduire un état sale.

12.6 Dépendance agent

Si l'agent est faible ou contournable,
la vision centrale devient trompeuse.

12.6 Dépendance agent

Si l'agent est faible ou contournable,
la vision centrale devient trompeuse.

Étape 1

Lab local ou VM de test.

Étape 2

Un petit lot de serveurs non critiques.

Étape 3

Reverse proxies secondaires.

Étape 4

Workers internes.

Étape 5

Services plus critiques.

Jamais :
déploiement massif d’un coup.

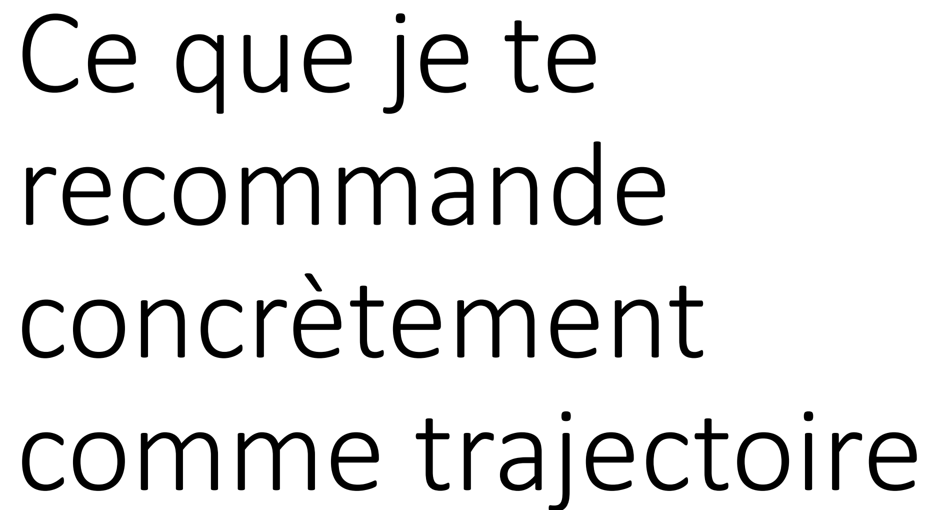
14. Gouvernance du projet

Rôles

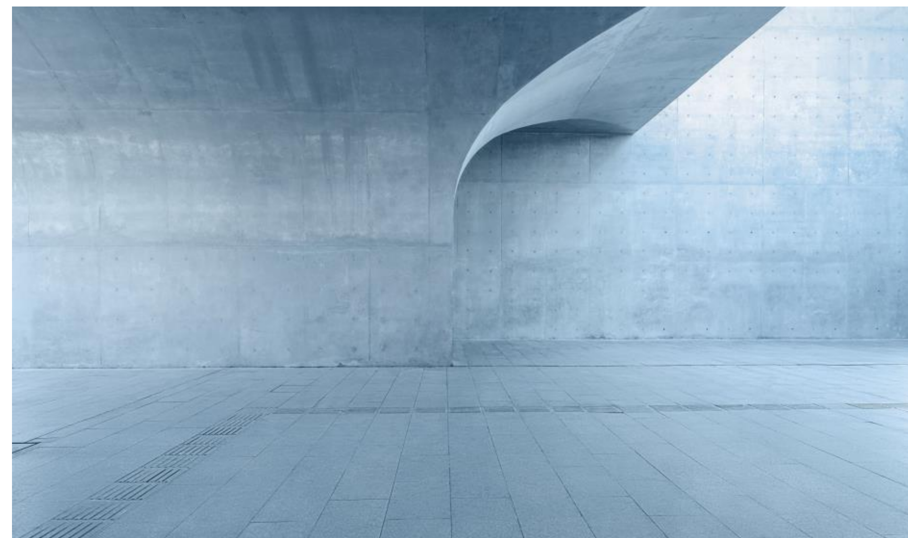
- architecte sécurité
- lead backend Django
- lead agent Linux
- ingénieur système Linux
- ingénieur cloud/rebuild
- ingénieur observabilité
- test/red team interne
- ops/support

Rituels

- revue threat model
- revue politiques
- revue incidents
- revue faux positifs
- revue post-rebuild
- revue de sécurité mensuelle



Ce que je te recommande concrètement comme trajectoire



V1

Créer le control plane Django et l’agent Python minimal.

V2

Ajouter policy engine, hardening, incidents, quarantine simple.

V3

Ajouter attestation logicielle et secrets éphémères.

V4

Ajouter rebuild automatisé.

Premier backlog de réalisation

Sprint A

- créer projet Django `aegis`
- apps de base
- modèle `Node`
- modèle `NodeHeartbeat`
- API enroll
- API heartbeat
- admin Django

Sprint B

- agent Python V1
- inventaire système
- systemd service
- communication API sécurisée
- dashboard flotte simple

Sprint C

- modèle `SecurityPolicy`
- fetch/apply policy
- hardening SSH/sysctl/services
- reporting de conformité

Sprint D

- événements sécurité
- incidents
- timeline nœud
- scoring simple

Sprint E

- soft quarantine
- nftables isolate
- blocage accès admin
- audit actions

Sprint F

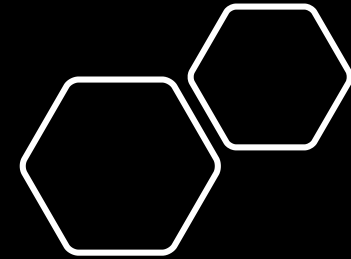
- attestation logicielle
- scoring de confiance
- trust states

Sprint G

- secrets éphémères V1
- révocation
- rotation

Sprint H

- recovery jobs
- rebuild orchestration
- historiques recovery



architecture final

- démarrer **sous Django**
- faire un **dashboard très pilotable**
- garder une **architecture modulaire**
- éviter dès le début la tentation “on va tout faire en Rust”
- d’abord produire une **plateforme qui fonctionne**, puis durcir
- tout concevoir avec une logique de **cron/management commands** + **dashboard** + **agents systemd**

Aegis Linux — Blueprint V1

- arborescence Django complète
- liste exacte des apps
- data model détaillé
- premiers endpoints API
- architecture de l’agent Linux
- premiers management commands
- structure du dashboard IDEO-Lab