Sécurisation des accès distants

Procédure de référence. Les adresses, noms d'hôtes et ports de cette page sont des exemples : les transposer à son propre parc.

Le fichier `.md` de cette page : ouvrir le `.md`

Ce que cette procédure couvre : inventorier ce qu'un routeur domestique expose réellement sur Internet, fermer ce qui n'a pas à l'être, durcir ce qui doit rester ouvert, et vivre avec au quotidien.

Le principe directeur : chaque port ouvert est une porte que quelqu'un frappera. La question n'est pas « est-ce risqué ? » mais « ce service a-t-il besoin d'être joignable depuis Internet, ou un tunnel suffit-il ? ». Presque toujours, un tunnel suffit.

> *Convention : chaque bloc indique où il s'exécute — `[routeur]`, `[nas-host]`, `[poste-bureau]`, `[téléphone]`, `[conteneur web]`.*

Le parc de référence

Machine	Adresse	Rôle
Routeur	192.168.0.1	Asus RT-AX86U sous Asuswrt-Merlin
nas-host	192.168.0.11	Hôte Incus + NAS, sans écran ni clavier
Conteneur web `web-srv`	192.168.0.6	ISPConfig + Apache, cible du 443 public
Pi-hole	192.168.0.5	DNS local
Beelink	192.168.0.106	Poste de travail
N73	192.168.0.105	Serveur Minecraft
WireGuard	10.6.0.0/24	Tunnel, serveur en 10.6.0.1

Domaine public : `example.com`. DDNS : `webadmin.asuscomm.com`.

Étape 1 — Inventorier ce qui est réellement exposé

Trois endroits à regarder dans le routeur, pas un seul.

La table de redirections — `[routeur]` → Réseau étendu → Redirection de port. Noter chaque ligne : nom, port externe, port interne, adresse interne, et surtout la colonne IP source, presque toujours vide.

⚠ Le port externe et le port interne peuvent différer. Une règle `8082 → 8080` fait qu'aucune commande lancée sur le serveur ne révélera le 8082 : il n'existe que dans la traduction du routeur.

La DMZ — `[routeur]` → Réseau étendu → DMZ. Un hôte en DMZ reçoit tout le trafic entrant et annulerait tout le reste du travail. Doit être désactivée.

L'UPnP — `[routeur]` → Réseau étendu → Connexion Internet. S'il est actif, n'importe quel logiciel du réseau peut ouvrir un port sans qu'il figure dans la table. L'inventaire est alors incomplet par construction.

Vérifier ensuite ce que le serveur écoute réellement :

```
[nas-host]
incus exec web-srv -- ss -tlnp | grep -E ':808'
```

Resserrer l'UPnP plutôt que le désactiver

⚠ Peu d'applications en dépendent réellement, et il faut vérifier plutôt que supposer. Un jeu en ligne ordinaire — World of Warcraft, ou un client Minecraft rejoignant un serveur — n'ouvre que des connexions sortantes : aucun port entrant, donc aucune demande au routeur. Un serveur Minecraft officiel ne parle pas UPnP non plus, d'où la redirection manuelle qu'il exige. Les outils de visioconférence courants — Teams, Zoom, Meet — n'en dépendent pas davantage :

leur média transite par les serveurs du fournisseur, ou traverse le NAT par STUN/TURN, qui perce un trou depuis l'intérieur au lieu de demander une redirection.

Les vrais utilisateurs sont plus rares qu'on ne croit :

- les clients BitTorrent ;
- les consoles de jeu, qui le réclament pour obtenir un « NAT ouvert » ;
- Syncthing, qui l'active par défaut sur le port 22000 ;
- Plex et Jellyfin, pour leur accès distant ;
- quelques serveurs de jeu auto-hébergés, via des mods.

La table `[routeur]` → Historique du système → Transfert de port tranche la question : elle liste les baux UPnP actifs. Vide alors que les applications tournent, rien n'en dépend — et l'UPnP peut être désactivé sans conséquence.

Si quelque chose en dépend, deux réglages le rendent acceptable sans rien casser :

- « Enable secure UPnP IGD mode » à Oui — une machine ne peut alors demander une redirection que vers elle-même, jamais vers une autre machine du réseau. C'est la protection qui compte le plus.
- Plage de ports externes de 1024 à 65535, et non de 1. Le `1` d'origine

> permet à une application de réclamer un port privilégié — 22, 443, 3389 —, précisément ceux qu'on vient de fermer.

Les baux existants ne sont pas affectés : la restriction ne s'applique qu'aux nouvelles demandes.

⚠ Vérifier ensuite que les applications qui dépendaient de l'UPnP fonctionnent toujours. Elles échouent en silence : un client BitTorrent privé de port entrant continue de télécharger, simplement moins bien connecté, et n'affiche rien d'autre qu'un indicateur de couleur.

Cette même page liste la table NAT complète — redirections manuelles et baux UPnP. C'est le seul endroit où l'on voit ce que l'UPnP a réellement ouvert.

Deux lignes `PCREDIRECT` sur le port 80 vers l'adresse du routeur lui-même y figurent normalement : c'est la page que le routeur affiche à la place d'un site quand Internet est tombé. Elle ne s'applique qu'au trafic local — vérifié : le port 80 ne répond pas depuis l'extérieur.

Cas particulier d'une machine derrière un VPN commercial. Si l'application est liée à l'interface du tunnel, sa demande UPnP part vers le VPN et n'atteint jamais le routeur : aucun bail n'apparaîtra, et c'est normal. Le port entrant se demande alors au fournisseur du VPN — Proton le fait par NAT-PMP sur ses serveurs P2P, avec un forfait payant, et son application maintient elle-même la redirection. Il ne reste qu'à donner ce port à l'application et à décocher son propre UPnP, pour qu'un seul acteur tienne le mappage. ⚠ Ce port change à chaque reconnexion.

Étape 2 — Décider quoi garder

Un seul critère : ce service doit-il être joignable par quelqu'un d'autre que moi ?

Réponse	Décision
Oui — sites web publics, serveur de jeu partagé	Garder
Non, mais j’y accède de l’extérieur	Tunnel VPN, pas de redirection
Non, et je n’y accède que de chez moi	Fermer

Décisions prises ici :

Service	Sort	Motif
HTTPS 443 → .6	Garder	Sert les sites publics
Minecraft 25565 → .105	Garder	Joueurs extérieurs
SSH 22 → nas-host	Garder, durci	Usage quotidien SolidExplorer ; Android n’autorise qu’un seul VPN à la fois
FTP 20,21 → .6	Fermer	Non chiffré ; le tunnel donne le même accès
FTP TLS 990 → .6	Fermer	Le tunnel suffit
SSH 8080 → .6:22	Fermer	Tunnel
SSH 8081 → .106:22	Fermer	Tunnel
Panneau ISPConfig 8082 → .6:8080	Fermer	Administration par tunnel
RDP 3389 → .105	Fermer	Premier vecteur des rançongiciels

⚠ Le RDP exposé est le point le plus grave qu’on puisse trouver dans une telle table. Le fermer passe avant tout le reste.

Étape 3 — Préparer avant de fermer

⚠ L’erreur classique : fermer d’abord et découvrir ensuite qu’on dépendait de la porte fermée.

Un nom public résolu vers l’IP publique fait sortir la requête même depuis le réseau local : elle atteint le routeur, qui la renvoie à l’intérieur. Fermer la règle coupe donc aussi l’accès local.

Un enregistrement DNS local — `[Pi-hole]` → Local DNS → DNS Records :

```
web-srv.example.com  →  192.168.0.6
```

Une ligne dans `/etc/hosts` sur toute machine dont le résolveur échappe à Pi-hole :

```
[poste-bureau]
echo "192.168.0.6 web-srv.example.com" | sudo tee -a /etc/hosts
getent hosts web-srv.example.com
```

⚠ Le contournement local d'un VPN commercial couvre le trafic, pas la résolution de noms. Sur le poste-bureau, Proton VPN laisse passer les adresses locales mais envoie les *noms* à son propre résolveur, qui répond l'adresse publique. `/etc/hosts` est consulté avant tout résolveur, VPN ou pas — c'est le seul correctif fiable.

Conséquence à connaître : tant qu'un VPN commercial gère la résolution d'une machine, Pi-hole ne filtre rien de ce que fait cette machine.

Éprouver le tunnel — `[téléphone]`, données cellulaires, WireGuard activé, en visant l'adresse et non le nom :

```
https://192.168.0.6:8080
```

Le nom ne fonctionnera que si le VPN pousse Pi-hole comme résolveur. L'adresse, elle, ne dépend de rien.

Étape 4 — Fermer

`[routeur]` → supprimer les règles décidées, puis Appliquer.

Noter les valeurs avant suppression de toute règle susceptible d'être rétablie :

```
Nom : RDP    Externe : 3389    Interne : 3389    IP : 192.168.0.105    TCP
```

La table de l’Asus n’offre pas d’interrupteur par règle — seulement Éditer et Supprimer.

Se faire depuis le réseau local : aucun risque de se couper l’accès au routeur.

Étape 5 — Vérifier depuis l’extérieur

⚠ Un test fait avec le VPN actif ne prouve rien : il passe par le tunnel, pas par la porte qu’on vient de fermer.

[téléphone], données cellulaires, VPN coupé :

- l’adresse fermée ne doit plus rien donner ;
- les sites publics doivent continuer de répondre.

 Un nom sans site dédié servi sur le 443 affiche le vhost par défaut d’Apache — chez nous, le blog. C’est normal et sans fuite.

Étape 6 — Durcir le SSH qui reste ouvert

Relever l’état effectif

```
[nas-host]
sudo sshd -T \
```

```
  | grep -E "^(port|permitrootlogin|passwordauthentication)"
```

`sshd -T` affiche la configuration effective — défauts et fichiers inclus compris — plutôt que ce qui est écrit dans `sshd_config`, qui peut être trompeur.

Le filet — accepter les mots de passe seulement de l'intérieur

⚠ Ne jamais interdire les mots de passe avant qu'une clé fonctionne. Sur une machine sans écran, l'erreur oblige à y brancher un clavier et un moniteur.

La forme ci-dessous supprime le risque au lieu de le gérer : elle ferme Internet et laisse le réseau local ouvert.

```
[nas-host]
sudo nano /etc/ssh/sshd_config.d/10-durcissement.conf
```

```
PasswordAuthentication no

Match Address 192.168.0.0/24,10.6.0.0/24
    PasswordAuthentication yes

Match all
```

Deux subtilités de `sshd_config`, inverses des habitudes d'Apache :

⚠ La première occurrence d'une directive l'emporte, pas la dernière. L'`Include /etc/ssh/sshd_config.d/*.conf` se trouvant en ligne 13 et le `PasswordAuthentication` d'origine en ligne 124, un fichier déposé dans `sshd_config.d/` gagne sans qu'on touche au fichier principal — que les mises à jour du paquet peuvent remplacer.

⚠ `Match all` en dernière ligne est indispensable. Un bloc `Match` s'applique à tout ce qui le suit jusqu'au `Match` suivant, et l'inclusion est textuelle : sans lui, le bloc s'étendrait sur tout le reste de `sshd_config`.

Valider avant d’appliquer

```
[nas-host]
sudo sshd -t
sudo sshd -T | grep -i passwordauthentication
sudo sshd -T -C addr=192.168.0.106,user=hostadmin,host=poste-bureau \
  | grep -i passwordauthentication
```

Attendu : silence, `no`, `yes`.

`sshd -T -C` évalue les blocs `Match` pour un contexte donné. C’est le seul moyen de prouver qu’un filet existe *avant* d’en avoir besoin. Aucune de ces commandes n’applique quoi que ce soit.

```
[nas-host]
sudo systemctl reload ssh
```

Ouvrir ensuite une seconde session sans fermer la première : une session établie survit à un rechargement fautif, seule une nouvelle connexion dit la vérité.

Une clé par appareil

Modèle : une clé par appareil, jamais partagée, nommée d’après l’appareil qui la détient. Ce qui doit être conservé n’est pas la clé privée mais `authorized_keys`, qui ne contient que des clés publiques et se sauvegarde sans précaution. Perdre un appareil se règle en retirant sa ligne ; les autres ne sont pas touchés.

```
[poste-bureau]
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519_pixel9 -C "Pixel9 solidexplorer" -N ""
```

`-N ""` crée la clé sans phrase de passe : un client mobile la redemanderait à chaque connexion et l’outil finirait par être abandonné. Compromis assumé — cette clé vaut par sa révocabilité.

Le carré de caractères affiché par `ssh-keygen` est le randomart, une représentation visuelle de l'empreinte. Purement informatif : rien à copier, rien à noter.

```
[poste-bureau]
ssh-copy-id -i ~/.ssh/id_ed25519_pixel9.pub hostadmin@192.168.0.11
ssh -i ~/.ssh/id_ed25519_pixel9 -o PasswordAuthentication=no \
  hostadmin@192.168.0.11 "echo cle pixel9 OK"
```

⚠ Le `-o PasswordAuthentication=no` de la seconde ligne est essentiel. Sans lui, une clé défaillante ferait retomber sur le mot de passe, la connexion réussirait, et on croirait la clé valide.

Le poste de travail : phrase de passe et agent SSH

⚠ Une clé par appareil vaut aussi pour le poste depuis lequel on administre. Créer les clés des appareils mobiles *sur* le poste ne donne aucune clé au poste : il continue de demander le mot de passe à chaque connexion.

Et le compromis s'y inverse. Sur un mobile, une phrase de passe serait redemandée à chaque connexion et l'outil finirait abandonné — d'où le `-N ""`. Sur un poste de travail, l'agent SSH retient la phrase pour toute la durée de la session : on la saisit une fois à l'ouverture, plus jamais ensuite. C'est donc le meilleur des deux, et la phrase se justifie pleinement.

 Noter la phrase de passe dans le gestionnaire de mots de passe avant de la taper, pas après. Elle n'existe nulle part ailleurs : ni le fichier de clé ni aucun outil ne la restituent. Une clé dont la phrase est perdue est une clé morte, et il ne reste qu'à en fabriquer une autre.

```
[poste]
ssh-keygen -t ed25519 -C "poste vers serveur"
ssh-copy-id utilisateur@192.168.0.11
```

Accepter le chemin par défaut (`~/.ssh/id_ed25519`) : c'est celui que SSH cherche

seul, donc rien à préciser ensuite ni à `ssh-copy-id`, ni à la connexion.

À la première connexion, l'invite doit demander `Enter passphrase for key …` et non `…'s password:` — c'est ce qui distingue une clé qui fonctionne d'un mot de passe qui persiste. À la seconde, plus rien n'est demandé.

Ce comportement se pose, il ne s'espère pas du bureau. Une ligne dans la configuration du client SSH, avant tout bloc `Host`, le garantit :

```
[poste]
nano ~/.ssh/config
```

À écrire en tête du fichier ouvert ci-dessus :

```
AddKeysToAgent yes
```

SSH charge alors la clé dans l'agent après le premier déverrouillage réussi. La phrase est demandée une fois par session, jamais deux, et rien n'est écrit sur le disque.

Quand l'agent du bureau annonce les clés sans savoir les utiliser

 Le trousseau de session peut se présenter comme agent SSH tout en étant incapable de signer. Son composant SSH énumère les clés trouvées dans `~/.ssh` sans les avoir chargées : `ssh-add -l` les affiche donc, empreinte comprise, et tout paraît en ordre — mais la connexion échoue sur `agent refused operation`.

Le piège est que la liste ressemble à une preuve. Elle n'en est pas une ; la seule qui vaille est une demande de signature.

```
[poste]
echo $SSH_AUTH_SOCK
ssh-add -T ~/.ssh/id_ed25519.pub
```

La première ligne dit quel agent répond. La seconde réclame une signature réelle avec cette clé précise — l'opération exacte qui échoue à la connexion, isolée du reste. Un refus ici, alors que `ssh-add -l` affichait la clé, désigne le trousseau et non la clé.

 Et `ssh` n'ouvre pas le fichier de clé pour autant. Dès que l'agent prétend détenir la clé nommée par `IdentityFile`, `ssh` la lui confie — les traces le disent en toutes lettres, `explicit agent` — puis, devant le refus, passe directement au mot de passe sans jamais essayer le fichier. La phrase de passe n'est donc pas demandée, et `AddKeysToAgent yes` n'a aucune occasion d'agir.

C'est ce qui rend la panne déroutante : la clé est bonne, le serveur l'accepte — les traces montrent `Server accepts key` —, et la connexion retombe quand même sur le mot de passe.

Le correctif : son propre agent, désigné explicitement

Discuter avec le trousseau ne mène nulle part : son composant SSH gagne la variable d'environnement au démarrage de session, et rien ne garantit l'ordre. Charger la clé à la main à chaque session n'est pas une réponse non plus — c'est un geste à retenir, donc un geste qu'on oubliera.

La réponse tient en deux pièces : un agent SSH ordinaire dans la session, et une configuration qui nomme sa prise plutôt que de s'en remettre à la variable.

```
[poste]
nano ~/.config/systemd/user/ssh-agent.service
```

Contenu à coller dans l'éditeur ouvert ci-dessus :

```
[Unit]
Description=Agent SSH de la session
```

```
[Service]
Type=simple
ExecStart=/usr/bin/ssh-agent -D -a %t/ssh-agent.socket

[Install]
WantedBy=default.target
```

`-D` garde l'agent au premier plan, comme systemd l'attend d'un `Type=simple`. `-a` lui impose l'emplacement de sa prise au lieu du chemin aléatoire qu'il choisirait seul — c'est ce qui rend l'adresse prévisible, donc citable dans une configuration. `%t` est résolu par systemd en `/run/user/<identifiant>`.

```
[poste]
systemctl --user daemon-reload
systemctl --user enable --now ssh-agent.service
systemctl --user is-active ssh-agent.service
ls -l /run/user/1000/ssh-agent.socket
```

Attendu : `active`, puis une prise `srw-------` à ce chemin.

Il reste à désigner cet agent dans la configuration du client, en tête du fichier, avant tout bloc `Host` :

```
IdentityAgent /run/user/1000/ssh-agent.socket
```

 `IdentityAgent` l'emporte sur la variable d'environnement. Peu importe désormais qui gagne la course au démarrage de session : `ssh` ne s'adressera qu'à cet agent, qui sait signer. Le trousseau continue de servir les mots de passe des applications, mais il ne touche plus au SSH.

⚠ Le chemin porte l'identifiant numérique de l'utilisateur, que `id -u` donne. C'est acceptable dans un fichier personnel — un agent SSH l'est par nature — mais ça interdit de recopier cette ligne telle quelle dans une configuration commune à plusieurs comptes.

Le contrôle qui tranche est une session neuve : ouvrir une session, lancer une connexion, et vérifier que la phrase de passe est demandée dans le terminal. Elle ne doit plus l'être ensuite, et le comportement doit se reproduire au démarrage suivant. Un correctif qui ne survit pas à un redémarrage n'est pas un correctif.

Transférer la clé au client mobile

SolidExplorer accepte le collage direct du texte de la clé, aussi bien qu'un fichier. Le collage évite tout transfert de fichier.

```
[poste-bureau]
cat ~/.ssh/id_ed25519_pixel9
```

Copier tout le bloc, lignes `-----BEGIN` et `-----END` incluses — elles font partie de la clé.

Le déposer dans une note sécurisée du gestionnaire de mots de passe, nommée `Clé SSH privée — Pixel 9 → nas-host`, avec la date et l'utilisateur en commentaire. Le coffre se synchronise vers le téléphone : la note sert à la fois de transfert et de sauvegarde.

`[téléphone]` → SolidExplorer → nouvelle connexion SFTP → « Clé privée SSH » → coller. Utilisateur `hostadmin`.

□□ Proton Drive et Proton Pass sont tous deux chiffrés de bout en bout. La distinction n'est pas la force du chiffrement mais le nombre de copies en clair sur disque : un fichier synchronisé existe en clair sur chaque machine qui le synchronise, et dans les sauvegardes de ces machines. Un secret dans un coffre ne se matérialise nulle part.

⚠□ Vérifier qu'aucun outil de synchronisation n'a `~/.ssh` dans son périmètre.

La clé privée peut ensuite être effacée de la machine qui l'a créée, le coffre en étant le dépôt :

```
[poste-bureau]
rm ~/.ssh/id_ed25519_pixel9
```

Une clé par appareil, pas par application

Toutes les applications d’un même appareil partagent la même clé. SolidExplorer et FolderSync sur le Pixel 9 utilisent tous deux `id_ed25519_pixel9` — il n’y a pas de nouvelle paire à générer pour chaque outil. C’est ce qui garde `authorized_keys` lisible et la révocation simple : une ligne retirée bannit l’appareil entier.

Deux formes de transfert selon le client

Les clients mobiles n’acceptent pas la clé de la même façon, et ça change la méthode.

Le client accepte le collage du texte — SolidExplorer, par exemple. C’est le cas simple : la note du gestionnaire de mots de passe suffit, rien ne transite en fichier.

Le client exige un fichier — FolderSync, par exemple. Il faut alors obtenir la forme fichier de la clé. Trois voies, par ordre de préférence :

1. Pièce jointe dans le gestionnaire de mots de passe, si l’offre le permet. Le fichier exact est joint à la note, téléchargé sur l’appareil, et désigné par l’application. Transit et conservation au même endroit, sans risque de déformation.

2. Câble USB entre le poste et l’appareil. Aucun risque de déformation, aucune copie résiduelle.

3. Recréer le fichier sur l’appareil en collant le texte de la note dans un éditeur. ⚠ Certains éditeurs Android ajoutent une extension `.txt` ou

> convertissent les fins de ligne en CRLF — et une clé en CRLF est refusée sans message clair. À vérifier après coup si la connexion échoue sans raison apparente.

 Ne pas passer par un dossier synchronisé pour ce transfert : la clé privée se retrouverait en clair sur chaque machine synchronisée et dans leurs sauvegardes.

 Si une clé privée revient un jour sur une machine Linux, ses droits doivent être resserrés — SSH refuse d'utiliser une clé lisible par d'autres :

```
[la machine concernée]
chmod 600 ~/.ssh/id_ed25519_pixel9
```

Sur Android la question ne se pose pas : chaque application dispose de son espace propre.

Remplacer une clé du poste

Une phrase de passe perdue, une clé compromise, un algorithme à renouveler : le remplacement suit le même ordre, et l'ancienne ne se retire qu'en dernier. Tant qu'elle reste autorisée, une erreur ne coûte rien.

1. Fabriquer la nouvelle paire à côté de l'ancienne, sous un nom distinct.
2. L'installer sur le serveur — les deux coexistent dans `authorized_keys`.
3. L'éprouver.
4. Inventorier `authorized_keys`.

5. Retirer la ligne de l’ancienne, après vérification.

L’agent fait obstacle à la deuxième étape. `ssh-copy-id` commence par tenter une connexion avec les clés de l’agent, qui propose la morte ; le trousseau réclame alors une phrase qu’on n’a plus. Le court-circuit tient dans le préfixe :

```
[poste]
SSH_AUTH_SOCK= ssh-copy-id -i ~/.ssh/id_ed25519_nouvelle.pub \
  utilisateur@192.168.0.11
```

Éprouver ensuite, sans laisser le mot de passe sauver la mise :

```
[poste]
SSH_AUTH_SOCK= ssh -i ~/.ssh/id_ed25519_nouvelle \
  -o IdentitiesOnly=yes -o PasswordAuthentication=no \
  utilisateur@192.168.0.11 "echo nouvelle cle OK"
```

Inventorier avant de couper. `authorized_keys` mélange les appareils, et une ligne retirée au jugé bannit le mauvais. Ce relevé n’affiche jamais les clés, seulement leur rang, leur empreinte et leur commentaire :

```
[serveur]
n=0; while IFS= read -r l; do n=$((n+1)); \
  printf "%s" "$l" > /tmp/k.$$; \
  echo "$n  $(ssh-keygen -lf /tmp/k.$$ 2>/dev/null)"; \
  done < ~/.ssh/authorized_keys; rm -f /tmp/k.$$
```

Supprimer par numéro de ligne sans vérifier l’empreinte est une erreur silencieuse : le fichier a pu changer entre le relevé et la coupe. La forme ci-dessous refuse d’elle-même si la ligne visée n’est pas celle attendue, et laisse une copie de sûreté :

```
[serveur]
```

```
cd ~/.ssh && cp -p authorized_keys authorized_keys.avant-menage && \
printf "%s" "$(sed -n 7p authorized_keys)" > /tmp/v.$$ && \
ssh-keygen -lf /tmp/v.$$ | grep -q EMPREINTE_DE_L_ANCIENNE && \
sed -i -e 7d authorized_keys && echo "ligne 7 retiree" \
|| echo "REFUS : rien n a ete fait"; rm -f /tmp/v.$$
```

Les fichiers de l'ancienne clé se renomment plutôt qu'ils ne se suppriment : un suffixe suffit à ce que SSH cesse de les regarder.

□□ Le poste est le seul appareil dont la clé ne se remplace pas en retirant sa ligne. Pour un mobile perdu, la ligne retirée règle tout. Pour le poste, c'est l'accès d'administration qui disparaîtrait — d'où cet ordre, et le mot de passe encore accepté depuis le réseau local qui sert de filet.

Traduire le port externe

`[routeur]` → modifier la règle SSH : externe 52200, interne 22.

⚠□ Ce n'est pas de la sécurité — un balayage complet trouve le port en quelques minutes. Mais les robots ne visent que le 22, et le mot de passe est déjà refusé : le port n'est plus un enjeu de protection, seulement de propreté des journaux.

□ Et sur ce terrain-là, déplacer le port bat `fail2ban` : l'un évite le bruit, l'autre le traite après coup. La mesure le confirme — après ce déplacement, le journal d'authentification de la machine ne contenait qu'une seule tentative ratée en une semaine. C'est pourquoi `fail2ban`, décrit à l'étape suivante, n'est posé qu'ensuite et pour d'autres raisons : il ne travaillera presque jamais, et c'est voulu.

La modification est entièrement au routeur : le port interne reste 22, la configuration du serveur ne bouge pas, l'accès local est intact.

Coût : un champ à corriger dans le client sur chaque appareil.

Éprouver la chaîne complète

`[téléphone]`, données cellulaires, VPN coupé. C’est le seul test qui exerce la redirection, le refus du mot de passe et l’acceptation de la clé.

Étape 6 bis — `fail2ban`, le filet sous le filet

À poser en dernier, et à comprendre pour ce qu’il est. Les trois gestes précédents — fermer les redirections inutiles, refuser le mot de passe hors du réseau local et du tunnel, déplacer le port — ont déjà vidé la menace de sa substance. Une force brute contre ce SSH ne peut plus aboutir : il n’y a plus de mot de passe à deviner.

 `fail2ban` n’est donc pas la défense, il en est l’assurance. Trois raisons de le poser quand même, et aucune n’est celle qu’on cite d’habitude :

- il bannit ce qui martèle, ce qui garde le journal lisible pour les rares événements qui comptent ;
- il protège le jour où le durcissement serait défait — une directive écrasée par une mise à jour du paquet, une clé retirée par mégarde ;
- il rend l’état visible, en portant deux compteurs qu’un tableau de bord peut relever.

⚠ Ne pas lui confier le rôle de première ligne. Un parc qui compte sur `fail2ban` pour protéger un SSH acceptant les mots de passe depuis Internet est un parc mal configuré avec un filet — pas un parc sécurisé.

Installer et configurer

[nas-host]

```
sudo apt install fail2ban
```

⚠ Le paquet démarre le service avec ses réglages d’origine, où seule la boucle locale est blanchie. Entre l’installation et la configuration, cinq échecs d’authentification depuis le réseau local suffiraient à s’auto-bannir. Enchaîner les deux gestes.

[nas-host]

```
sudo nano /etc/fail2ban/jail.local
```

```
[DEFAULT]
ignoreip  = 127.0.0.1/8 ::1 192.168.0.0/24 10.6.0.0/24
backend   = polling
banaction = iptables-multiport
bantime   = 3600
findtime  = 600
maxretry  = 5

[sshd]
enabled  = true
port     = ssh
logpath  = /var/log/auth.log
maxretry = 5
findtime = 600
bantime  = 86400
```

 `ignoreip` doit reprendre exactement les plages du `Match Address` de `sshd_config`, tunnel compris. Les deux listes décrivent la même notion — « d’où je suis chez moi » — et doivent rester identiques. Une plage ajoutée à l’une et oubliée dans l’autre donne un accès qui fonctionne jusqu’au cinquième essai raté, puis disparaît sans explication.

⚠️ Oublier le sous-réseau du tunnel est l'erreur classique, parce qu'on pense « réseau local » et qu'on écrit la plage du domicile. Une administration menée à distance par le tunnel arrive avec une adresse `10.6.0.x` : sans elle, elle est traitée comme n'importe quel inconnu d'Internet.

`port = ssh` désigne le port interne, 22 — celui que voit le serveur. La traduction du routeur est invisible d'ici, et c'est bien le 22 qu'il faut bloquer.

`bantime` plus long pour SSH que par défaut : le blanchiment rendant l'auto-bannissement impossible depuis le domicile ou le tunnel, rien ne plaide pour un bannissement court. Et puisque le but est la propreté du journal, un balayeur écarté pour une journée sert mieux qu'un balayeur qui revient dans l'heure.

Vérifier sur le service, pas sur le fichier

[nas-host]

```
sudo systemctl restart fail2ban
sudo fail2ban-client status sshd
sudo fail2ban-client get sshd ignoreip
```

 La dernière commande est la seule qui prouve quelque chose. Elle interroge le service et non le fichier : un `jail.local` mal placé, mal nommé ou mal lu laisse un fichier d'apparence parfaite et un service qui applique ses réglages d'origine. La liste rendue doit contenir les quatre entrées, tunnel compris.

Les prisons du serveur web, et ce qu'elles surveillent vraiment

Le conteneur web porte ses propres prisons, et l'une d'elles illustre un piège qui vaut pour toutes.

 Un filtre `fail2ban` surveille une adresse, pas une fonction. Le filtre livré pour WordPress cherche `POST /wp-login.php` dans le journal d'Apache. Renommer la page de connexion — ce que toute bonne pratique recommande — rend donc ce

filtre aveugle du jour au lendemain, sans un message, sans un compteur qui change d’allure. La prison continue d’afficher des échecs et des bannissements : ceux des robots qui frappent encore l’ancienne porte.

⚠️ Toute mesure de sécurité qui déplace ou renomme quelque chose demande donc de relire ce qui surveillait l’ancien nom. Le renommage et la surveillance se configurent à deux endroits sans rapport l’un avec l’autre, et rien ne les relie.

Le renommage crée en revanche une occasion. L’ancienne adresse ne peut plus recevoir un seul visiteur légitime : tout ce qui y poste est un robot, sans exception possible. Elle devient un pot de miel, où le bannissement se justifie dès le premier essai.

Prison	Ce qu’elle vise	Essais tolérés
`wordpress-login`	la vraie page de connexion, échecs seulement	5
`wordpress-honeypot`	l’ancienne adresse et `xmlrpc.php`, devenus pièges	1
`apache-401`	les refus d’authentification HTTP	3

 Le filtre de la vraie page ancre sur le code de réponse, et c’est ce qui le rend juste. Une tentative ratée redonne le formulaire — 200 ; une réussie redirige vers l’administration — 302. En exigeant le `200`, la prison ne compte que les échecs, là où le filtre d’origine comptait aussi les connexions réussies.

```
failregex = ^(?:\S+:\d+ )?<HOST> .*"POST /porte-privee/ HTTP/[^"]+" 200
```

⚠️ Ces deux codes se vérifient, ils ne se supposent pas. Une extension de sécurité peut très bien rediriger sur échec plutôt que redonner le formulaire, ce qui inverserait complètement la règle. Une tentative volontairement ratée puis une réussie, suivies d’un coup d’œil au journal, tranchent en deux minutes.

Et le filtre s’éprouve sans rien redémarrer :

```
fail2ban-regex /var/log/apache2/other_vhosts_access.log \
  /etc/fail2ban/filter.d/wordpress-login.conf
```

Il rend le nombre de lignes retenues. Une seule, correspondant à l’essai raté et pas à l’essai réussi, prouve la règle mieux qu’un redémarrage suivi d’une attente.

 Et c’est le seul outil qui prouve quelque chose, parce qu’un compteur à zéro ne veut rien dire. Après un changement de filtre, la prison peut afficher zéro échec alors que le journal contient des lignes qu’elle devrait retenir : `fail2ban` reprend sa lecture à la position mémorisée dans sa base et ne compte que ce qui tombe dans la fenêtre de `findtime`. Un zéro peut donc signifier « rien à signaler » aussi bien que « mon expression est fausse » — et les deux se ressemblent parfaitement. `fail2ban-regex` relit le fichier entier sans tenir compte ni de la position ni de la fenêtre : c’est lui qui tranche.

Le filtre du pot de miel, pour mémoire :

```
failregex = ^(?:\S+:\d+ )?<HOST> .*"POST /(?:wp-login\.php|xmlrpc\.php)
```

⚠ Sur les `POST` seulement, jamais les `GET`. Un robot qui tente d’entrer poste ; un moteur de recherche suivant un vieux lien ne fait que demander la page. Bannir sur un `GET` finirait par écarter l’indexation du site.

Fermer plutôt que surveiller : le cas de `xmlrpc.php`

Cette interface de publication à distance est une cible constante, et elle n’a d’utilité que pour l’application mobile de WordPress, Jetpack, et quelques outils de publication. Si aucun de ceux-là n’est employé, la surveiller est le mauvais réflexe : il faut la fermer.

```
<Location /xmlrpc.php>
    Require all denied
```

```
</Location>
```

Même technique que le `<Location>` de l’étape 7 — un fichier dans `conf-available/`, activé par `a2enconf`, précédé d’un `apache2ctl configtest`.

 C’est le principe directeur de cette page appliqué à l’intérieur du serveur : la question n’est pas « comment protéger ce service », mais « ce service a-t-il besoin d’être joignable ». Une porte fermée ne se surveille pas.

⚠ Vérifier l’usage avant de fermer, et dans le bon ordre. Retirer d’abord l’application mobile, fermer ensuite. Le journal tranche : un seul `POST /xmlrpc.php` sur plusieurs milliers de lignes est le profil d’une sonde, pas d’un usage.

 Une fois fermée, l’adresse rejoint le pot de miel — et les deux mesures se renforcent. Le `403` empêche l’exploitation ; le bannissement écarte celui qui a essayé, pour que le journal reste lisible. Fermer sans bannir laisse les robots frapper indéfiniment ; bannir sans fermer laisse une porte que seul le gardien protège.

Vérifier la fermeture, et surtout l’absence de dégât collatéral :

```
curl -sk -o /dev/null -w "%{http_code}\n" -X POST \
  -H "Host: blog.example.com" https://127.0.0.1/xmlrpc.php
curl -sk -o /dev/null -w "%{http_code}\n" \
  -H "Host: blog.example.com" https://127.0.0.1/
```

Attendu : `403` puis `200`. ⚠ La seconde ligne n’est pas de la politesse : un `<Location>` mal ciblé ferme plus large que prévu, et ça ne se verrait qu’à la prochaine visite d’un lecteur.

```
grep -c 'POST /xmlrpc.php' /var/log/apache2/other_vhosts_access.log
```

Mesurer plutôt que fermer, quand la fermeture coûterait un usage

Six sites sont servis par le 443 public, et cinq sont gardés : le blog est public par vocation, la galerie demande un mot de passe au niveau d’Apache, la diffusion musicale et l’administration du routeur ont chacune leur authentification. Le sixième — un correcteur linguistique auto-hébergé — accepte et traite du texte sans aucune authentification.

 Le critère qui sépare les six n’est pas « site de contenu contre proxy inverse », c’est « authentifié ou non ». La distinction paraît évidente une fois posée ; elle ne l’est pas au moment de dresser la liste, où l’on classe spontanément par nature technique plutôt que par ce qui protège.

Fermer ce service au monde serait le geste cohérent avec le reste de cette page. Il ne l’est pas ici, parce qu’on ne sait pas d’avance si on en aura besoin de l’extérieur, et qu’un service fermé un jour de besoin est un service qu’on rouvre à la hâte, mal.

La réponse est alors de mesurer. Le journal d’Apache donne l’information exacte : combien de requêtes ce site a-t-il reçues, hier, d’ailleurs que du réseau local et du tunnel.

```
HIER=$(LC_ALL=C date -d yesterday +%d/%b/%Y)
JX=/var/log/apache2/other_vhosts_access.log
grep -h "\[$HIER" $JX $JX.1 2>/dev/null \
  | awk '$1 ~ /^lt.example.com:/ &&
         $2 !~ /^(192\.168\.0\.|10\.6\.0\.|127\.0\.0\.1)/' \
  | wc -l
```

⚠ `LC_ALL=C` n’est pas cosmétique. Apache écrit ses dates en anglais quelle que soit la langue du système ; sans ce réglage, `date +%b` rendrait « sept. » et ne correspondrait à rien.

⚠ L’adresse du routeur compte comme locale, et c’est correct — mais cela veut dire qu’aucune machine du réseau ne se distingue d’une autre. Le retour en

épingle masque la machine d’origine derrière l’adresse du routeur dès qu’on vise le nom public depuis la maison.

 Porté au tableau de bord, ce compte devient une décision différée plutôt qu’un pari. Il affiche zéro, et affichera zéro tant que personne n’aura trouvé l’adresse. Le jour où ce n’est plus zéro, la restriction se décide sur un fait — le lendemain, pas au centième visiteur.

⚠ Et le zéro doit rester affiché, pas seulement déclencher une alerte. Une mesure qui ne parle que lorsqu’elle se déclenche est indiscernable d’une mesure morte.

Si la restriction devient nécessaire : le piège du `<Location>`

 `<Location>` s’applique par CHEMIN, pas par hôte. Un `<Location /phpmyadmin>` déposé dans `conf-available/` fonctionne parce que ce chemin n’existe que là. Mais un site servi à la racine n’a pas de chemin qui lui soit propre : un `<Location />` global s’appliquerait à tous les sites du serveur à la fois, y compris ceux qui doivent rester publics.

Restreindre un site entier demande une directive de la portée du site, pas du serveur. Dans un panneau d’hébergement, c’est le champ *Directives Apache* de la configuration du site — stocké en base et réémis à chaque régénération, donc il survit là où un fichier édité à la main serait écrasé.

```
<Location />
    Require ip 192.168.0.0/24
    Require ip 10.6.0.0/24
</Location>
```

⚠ Deux techniques qui se ressemblent, dont l’une ferme un chemin et l’autre fermerait tout le serveur. C’est le genre de différence qu’on ne voit qu’après.

Le laisser lire par un relevé automatique

Le socket de `fail2ban` est en `0700` root : aucun réglage de permissions ne le rend lisible par un compte de service. Si un tableau de bord doit relever ces compteurs, la seule voie est une règle `sudo` nommée au plus étroit — une commande exacte, en lecture seule, sans joker.

[nas-host]

```
sudo visudo -f /etc/sudoers.d/tableau-de-bord
```

```
hostadmin ALL=(root) NOPASSWD: /usr/bin/fail2ban-client status sshd
```

⚠ Ne jamais éditer un fichier de `/etc/sudoers.d/` avec un éditeur ordinaire. Une erreur de syntaxe y rend `sudo` inutilisable sur toute la machine, y compris pour la réparer. `visudo` refuse d'enregistrer un fichier invalide — c'est la seule raison de l'employer, et elle suffit.

Le nom du fichier ne contient pas de point : comme pour `/etc/cron.d/`, `sudo` ignore en silence les fichiers dont le nom en comporte un.

Étape 7 — Les interfaces d'administration servies par alias global

Un panneau d'administration peut être public sans figurer dans aucune table de redirections, s'il est servi par le port 443 qu'on a décidé de garder.

Cas rencontré : `https://blog.example.com/phpmyadmin` répondait depuis le réseau cellulaire. Ce n'était pas un lien dans le site, mais un alias Apache global installé par le paquet, qui répond sous *chaque* vhost.

```
[nas-host]
incus exec web-srv -- grep -Rni "Alias /phpmyadmin" \
  /etc/apache2/ /etc/phpmyadmin/
```

⚠ `grep -r` ne suit pas les liens symboliques rencontrés pendant la descente ; `-R` les suit. Sur Debian, `conf-enabled/` n'est fait que de liens vers `conf-available/`, qui pointe lui-même ailleurs. Une recherche `-r` manque donc le fichier en silence — elle ne renvoie rien, ce qui se lit à tort comme « ça n'existe pas ».

Le correctif : un bloc `<Location>`.

```
[nas-host]
incus exec web-srv -- nano \
  /etc/apache2/conf-available/zzz-phpmyadmin-restrict.conf
```

```
<Location /phpmyadmin>
    Require ip 192.168.0.0/24
    Require ip 10.6.0.0/24
</Location>
```

⚠ `<Location>` l'emporte sur `<Directory>`. Apache applique les sections par type — `<Directory>` d'abord, `<Location>` en dernier — quel que soit l'ordre des fichiers. C'est ce qui permet de passer outre un `Require all granted` posé par un autre paquet sur le même répertoire, sans toucher à un fichier qu'il régénère.

Le préfixe `zzz-` n'a aucun effet technique : c'est un repère de lecture.

```
[nas-host]
incus exec web-srv -- a2enconf zzz-phpmyadmin-restrict
incus exec web-srv -- apache2ctl configtest
```

⚠ `configtest` avant tout rechargement, sans exception : ce serveur porte tous les sites. L'avertissement `AH00548: NameVirtualHost has no effect` est une mention

de dépréciation permanente, sans conséquence.

```
[nas-host]
incus exec web-srv -- systemctl reload apache2
```

Ne pas désactiver l’alias avec `a2disconf` : il sert aussi le vhost d’administration interne, et le retirer coupe l’accès de partout, y compris depuis le réseau local.

Vérifier

```
[nas-host]
incus exec web-srv -- curl -sk -o /dev/null \
  -w "%{http_code}\n" -H "Host: blog.example.com" https://127.0.0.1/
```

Origine	Attendu
Machine du LAN	200
Cellulaire avec VPN	200
Cellulaire sans VPN	403
Boucle locale du serveur	403 — normale, hors des plages
Racine des sites	200 — rien n’est cassé

Usage au quotidien

Besoin	Chemin
Panneau ISPConfig, depuis le LAN	`https://web-srv.example.com:8080`
Panneau ISPConfig, de l’extérieur	WireGuard puis `https://192.168.0.6:8080`
phpMyAdmin	`https://blog.example.com/phpmyadmin` — LAN ou VPN
Fichiers sur nas-host, mobile	SolidExplorer, `webadmin.asuscomm.com:52200`, clé
Administration en ligne de commande	WireGuard puis `ssh hostadmin@192.168.0.11`

⚠ Android n’autorise qu’un seul VPN actif à la fois. Un VPN commercial et

WireGuard s’excluent sur le même téléphone — c’est ce qui justifie de garder une redirection SSH durcie plutôt que de tout faire passer par le tunnel.

Les pièges, rassemblés

Le port externe peut différer du port interne. Aucune commande lancée sur le serveur ne révélera le port externe.

Un navigateur masque le préfixe `https://` et affiche « Non sécurisé » aussi bien pour du HTTP que pour du HTTPS à certificat accepté manuellement. Ne jamais en conclure qu’un service est en clair : le vérifier.

```
[nas-host]
incus exec web-srv -- curl -sS -o /dev/null \
  -w "%{http_code}\n" http://127.0.0.1:8080/
incus exec web-srv -- curl -skS -o /dev/null \
  -w "%{http_code}\n" https://127.0.0.1:8080/
```

Un `400` en HTTP et un `302` en HTTPS signifient : le port parle TLS, et Apache répond qu’on lui parle en clair.

`PR_CONNECT_RESET_ERROR` sur un port qui devrait servir du HTTPS peut signifier que la requête atterrit ailleurs. Ici, une résolution vers l’IP publique menait au port 8080 externe, redirigé vers SSH : une poignée de main TLS contre un serveur SSH donne exactement ce message.

Un test avec VPN actif ne prouve rien d’un chemin sans VPN.

`grep -r` ne suit pas les liens symboliques ; `-R` oui.

`<Location>` l’emporte sur `<Directory>` dans Apache ; la première occurrence l’emporte dans `sshd_config`. Deux logiques inverses dans deux fichiers voisins.

Vérifier sur quelle machine on tape. Trois machines et un conteneur suffisent à

se tromper — une commande sans effet apparent est souvent une commande lancée au mauvais endroit.

Résultat

	Avant	Après
Redirections de port	9	3
Restreintes par IP source	0	—
Mots de passe SSH depuis Internet	acceptés	refusés
Clés SSH sur le parc	aucune	une par appareil
Panneau d’administration public	oui	non
phpMyAdmin public	oui	non
RDP public	oui	non
Bannissement automatique	aucun	`fail2ban`, réseau local et tunnel blanchis

 La ligne la plus instructive est la troisième. Refuser les mots de passe depuis Internet est ce qui a rendu toutes les autres possibles : sans mot de passe à deviner, le port n’est plus qu’une question de propreté, et le bannissement automatique n’est plus qu’une assurance. L’ordre des gestes compte plus que leur nombre.