ChatGPT

Authentification Codex-Serveur via Puzzle Asymétrique

Ce document décrit le processus d'authentification entre le client **Codex** et le **serveur**, qui repose sur un **puzzle asymétrique** et l'usage de l'API **DigitalOcean** pour échanger un token et des métadonnées. Le mécanisme utilise un **challenge chiffré** unique (le puzzle) et un token secret, ainsi que des communications via le nom et les métadonnées d'un **Droplet DigitalOcean**. Les différentes étapes sont détaillées ci-dessous.

1. Clé publique et token initial

L'authentification commence par la génération d'une paire de clés RSA asymétriques et d'un token secret aléatoire côté serveur. La **clé publique** du serveur est partagée avec le client (Codex) – elle est même stockée en clair côté client pour être utilisée dans le puzzle (hypothèse acceptable car c'est une clé publique). De son côté, le serveur conserve la **clé privée** associée, gardée secrète.

En parallèle, le serveur génère un **token** secret (une valeur aléatoire suffisamment complexe). Ce token servira de **solution au puzzle** asymétrique [1]. Il est essentiel que ce token ne soit connu initialement que du serveur.

Implémentation: Le dépôt comprend un script Lisp et Shell permettant de gérer ces clés. Par exemple, `sandbox_auth.lisp` génère une paire RSA 4096 bits pour chaque variable d'environnement `SECRET_...`, en stockant la clé privée localement et en exportant la clé publique prête à être partagée [2]. Le script Shell `setup-auth.sh` exécute cette génération et peut même enregistrer automatiquement les clés publiques via l'API DigitalOcean si un token API (`DO_API_TOKEN`) est fourni [3]. Ainsi, dès l'initialisation, le client dispose de la clé publique du serveur et le serveur a préparé un token secret pour le challenge.

2. Publication du puzzle sur DigitalOcean

Le serveur construit ensuite le **puzzle asymétrique** en utilisant la clé publique partagée et le token. Le puzzle est en quelque sorte un défi chiffré que seul le client pourra résoudre. Concrètement, le serveur **chiffre le token avec une clé asymétrique** ou génère un artefact cryptographique dérivé du token (par exemple un **chiffrement RSA** du token ou une **signature** incluant le token). L'objectif est que Codex puisse, à partir de cet artefact, retrouver le token original, prouvant ainsi son authenticité.

Une fois le puzzle construit (généralement sous forme de texte encodé en base64 ou hexadécimal), le serveur le **publie via l'API DigitalOcean**. Ici, DigitalOcean sert de canal d'échange de métadonnées entre le serveur (droplet) et le client. Par exemple, le serveur peut utiliser l'API REST de DigitalOcean pour **renommer le Droplet** courant ou mettre à jour un tag/attribut contenant les informations du puzzle. Le nom du droplet est alors structuré de façon à inclure les données nécessaires, typiquement :

- **Le puzzle chiffré** – une représentation du défi (par exemple une longue chaîne base64).
- Éventuellement un identifiant lié à la tâche ou un préfixe explicite (ex: `"puzzle-..."`).

Exemple (côté serveur) – Publication du puzzle en insérant le puzzle chiffré dans le nom du droplet via l'API DigitalOcean :

```
# Variables supposées définies : DO_API_TOKEN (token API DO), DROPLET_ID
(identifiant du droplet serveur)
encryptedPuzzle="<...PuzzleChiffréBase64...>"  # puzzle généré à partir du
token
curl -X PUT -H "Authorization: Bearer $DO_API_TOKEN" \
     -H "Content-Type: application/json" \
     -d "{\"name\":\"puzzle-$encryptedPuzzle\"}" \
     "https://api.digitalocean.com/v2/droplets/$DROPLET_ID"
```

Dans cet exemple, le serveur renomme le droplet en préfixant le puzzle par `puzzle-`. Le puzzle chiffré (très long) est ainsi déposé dans le nom du droplet (NB : il pourrait aussi être placé dans des **métadonnées utilisateur** ou **tags** du droplet selon les besoins, le nom étant ici l'un des canaux disponibles via l'API).

3. Récupération du puzzle côté client

Le client Codex, disposant de l'accès à l'API DigitalOcean (via le même `DO_API_TOKEN` sécurisé), interroge à son tour l'API pour récupérer les informations du droplet serveur. Il obtient ainsi le **nom du droplet** et y trouve le puzzle publié à l'étape précédente.

Exemple (côté client) – Récupération du puzzle via l'API :

```
# Récupérer les informations du droplet et extraire le champ "name"
droplet_info=$(curl -s -H "Authorization: Bearer $DO_API_TOKEN" \
                  "https://api.digitalocean.com/v2/droplets/$DROPLET_ID")
droplet_name=$(echo "$droplet_info" | jq -r '.droplet.name')
# Supposons droplet_name = "puzzle-BASE64ENCSTRING"
puzzle_cipher=$(echo "$droplet_name" | sed 's/^puzzle-//')
echo "Puzzle chiffré récupéré : $puzzle_cipher"
```

Le client extrait ainsi la portion correspondant au puzzle chiffré (ici `BASE64ENCSTRING`). À ce stade, Codex a récupéré le puzzle asymétrique publié par le serveur, et possède la clé publique du serveur nécessaire pour vérifier son authenticité ou sa structure.

4. Résolution du puzzle asymétrique (côté client)

À partir du puzzle récupéré, le client doit **résoudre le puzzle**, c'est-à-dire retrouver le token secret initial. Étant donné la nature asymétrique du challenge, seul le client légitime peut effectuer cette opération :

- Si le puzzle a été construit comme un **chiffrement RSA** du token avec la clé publique du client, alors Codex utilise sa **clé privée** pour déchiffrer le message et récupérer le token.
- Si le puzzle est une **signature** du token (ou d'une donnée le contenant) faite avec la clé privée du serveur, alors Codex utilise la **clé publique du serveur** pour vérifier la signature et ainsi valider que le token associé est correct et provient du serveur.

- D'autres variantes combinent chiffrement et signature : par exemple, le serveur peut chiffrer le token avec la clé publique du client **et** signer cet ensemble avec sa clé privée. Le client devra alors d'abord vérifier la signature (avec la clé publique serveur) puis déchiffrer le contenu (avec sa propre clé privée).

Dans tous les cas, Codex obtient finalement le **token en clair** – c'est la **solution du puzzle**, comme prévu [1]. À ce stade, l'agent (Codex) **vérifie la validité** de la solution obtenue, par exemple en contrôlant la signature ou tout autre indicateur d'authenticité. Si la solution est valide, le client est sûr que le puzzle était bien émis par le serveur et que le token est correct.

Il est impératif que Codex **détruise immédiatement le puzzle et le token en mémoire** une fois utilisés, afin d'éviter toute répétition ou fuite ultérieure [4]. Cette étape garantit qu'aucune autre tâche ne pourra réutiliser ces secrets (preuve de non-répétition).

5. Envoi de la solution et métadonnées d'accès

Une fois le token en main, le client va prouver au serveur qu'il l'a correctement obtenu. Pour ce faire, il utilise à nouveau l'API DigitalOcean afin de **transmettre la solution** (le token en clair, ou une preuve de possession de celui-ci) ainsi que des métadonnées nécessaires à l'établissement de la session.

Concrètement, le client peut à nouveau **renommer le droplet** ou définir des **tags** de la façon suivante :

- Inclure un **identifiant d'utilisateur temporaire** (généralement généré par le serveur ou dérivé du token) dans le nom du droplet.
- Inclure le **token solution** (ou un hash de celui-ci) dans le nom ou un champ convenu, pour le communiquer au serveur.

Par exemple, après résolution, Codex pourrait renommer le droplet ainsi : `"user-3e7a9f-${token}"` (où `user-3e7a9f` est le nom d'utilisateur temporaire hexadécimal proposé, et `${token}` le token en clair ou un identifiant lié).

Exemple (côté client) – Transmission de la solution via l'API DO :

```
temp_user="user-3e7a9f"  # identifiant utilisateur temporaire fourni ou
calculé
solution="$token"        # le token solution en clair
curl -X PUT -H "Authorization: Bearer $DO_API_TOKEN" \
     -H "Content-Type: application/json" \
     -d "{\"name\":\"$temp_user-$solution\"}" \
     "https://api.digitalocean.com/v2/droplets/$DROPLET_ID"
```

Ici, le droplet serait renommé en par exemple `user-3e7a9f-S3cr3tT0k3n`, ce qui combine l'identifiant d'utilisateur éphémère et la preuve de connaissance du token. Le choix du format (nom du droplet vs. tags) importe peu du moment que le serveur et le client s'accordent sur la manière de lire ces données via l'API.

6. Validation par le serveur et création de l'utilisateur éphémère

Le serveur (ou un script s'exécutant sur le droplet) va périodiquement interroger l'API DigitalOcean afin de détecter la réponse du client. Lorsqu'il voit apparaître dans le nom (ou les métadonnées) du droplet l'identifiant temporaire et le token attendu, il sait alors que le client a résolu le puzzle avec succès.

À ce stade, le serveur compare le token reçu avec le token initial qu'il avait gardé secret. Si les deux correspondent, la **solution est validée**. Le serveur peut alors finaliser l'authentification en **préparant l'accès** au système :

- Un **compte utilisateur temporaire** est créé sur le serveur, avec comme nom d'utilisateur le code hexadécimal transmis (ex: `user-3e7a9f`). Ce compte est non-root et possède uniquement les permissions strictement nécessaires à l'exécution de la tâche prévue.
- Le **mot de passe temporaire** de ce compte est défini comme étant le token solution validé. Ainsi, seul celui qui a résolu le puzzle connaît ce mot de passe.
- Le serveur peut également inscrire la clé publique du client (générée en étape 1) dans le fichier `authorized_keys` de cet utilisateur temporaire, permettant une connexion par clé SSH en plus/en alternative au mot de passe.

Techniquement, la création de l'utilisateur et la configuration de ses accès peuvent être réalisées via un script Shell côté serveur. Ce script peut être déclenché automatiquement **lorsque le mot de passe temporaire est utilisé** pour une tentative de connexion, ou dès que le serveur valide le token (selon l'implémentation choisie). Par exemple, le serveur pourrait utiliser un mécanisme PAM ou un **hook sur la tentative de login** pour détecter l'utilisateur inconnu avec le mot de passe correct, puis exécuter un script qui appelle `useradd` pour créer le compte à la volée et configure son mot de passe. Dans une implémentation plus simple, le serveur, après validation, peut directement exécuter la création du compte et l'ouverture des accès avant que le client ne se connecte.

L'utilisation d'un **script côté serveur déclenché par le mot de passe temporaire** offre l'avantage de ne créer le compte qu'au moment nécessaire et de le supprimer ensuite, améliorant la sécurité (pas de compte dormant). Dans tous les cas, une fois cette création effectuée, le serveur notifie (éventuellement via l'API ou simplement en étant prêt à accepter la connexion) que Codex peut se connecter.

7. Connexion du client et exécution de la tâche

Le client Codex peut maintenant **ouvrir la session SSH** vers le serveur en utilisant l'**identifiant temporaire** et le **mot de passe (token)**. Grâce aux étapes précédentes, le serveur autorise cette connexion : le compte existe et le mot de passe correspond. Codex accède ainsi au serveur avec des droits limités, uniquement ce compte utilisateur éphémère.

Une fois connecté, Codex effectue la **tâche prévue** en utilisant ce compte (par exemple, exécuter un script, déployer du code, etc.), conformément aux permissions minimales allouées [5]. Le tout se fait de manière sécurisée, car seul l'agent ayant résolu le puzzle possède les identifiants corrects.

8. Nettoyage et révocation de l'accès

Lorsque la tâche est terminée, le serveur procède au **nettoyage** de la session d'agent. Cela comprend :

- La **suppression du compte utilisateur temporaire** créé pour Codex (`user-3e7a9f` dans notre exemple). Ainsi, plus aucune connexion future avec cet identifiant n'est possible une fois la fenêtre de la tâche fermée.
- L'invalidation du token utilisé : puisque le token/puzzle était à usage unique, il n'est plus jamais accepté après coup. Le puzzle et le token ayant été détruits côté client et serveur, toute **trace du token initial est effacée** du système [6].
- Le droplet peut éventuellement être renommé pour retirer les informations de puzzle/solution des métadonnées, ou ces métadonnées peuvent être nettoyées via l'API.

En suivant ce protocole, **l'accès est éphémère et zero-trust** : chaque tâche requiert un nouveau puzzle/token, aucun secret n'est réutilisé ni stocké durablement, et l'**agent s'efface une fois le rite accompli** [6].

Exemple synthétique de puzzle chiffré

Pour illustrer le fonctionnement, voici un exemple simple de puzzle asymétrique **chiffré** et sa solution :

- **Token secret initial (solution attendue)** : `"S3cr3tT0k3n"` (exemple de token généré aléatoirement).
- **Clé publique du serveur (RSA)** : on utilise une clé RSA (ex. 2048 bits) dont la portion publique est partagée au client.
- **Puzzle généré** : un chiffrement RSA du token avec la clé publique du **client** (de sorte que seul le client puisse le déchiffrer), encodé en base64 pour le transport. Par exemple, on obtient un texte chiffré :

  ```
  Puzzle (base64) : mQENBFey...ABAQAB\n...==
  ```

  (tronqué ici pour lisibilité – il s'agit d'une longue chaîne résultant du chiffrement RSA du token).

- **Publication** : le serveur publie ce puzzle via DO (par ex. nom du droplet `puzzle-mQENBFey...==` comme montré précédemment).
- **Récupération et résolution** : le client extrait le puzzle chiffré, puis le déchiffre avec **sa clé privée** (car dans cet exemple, on a considéré que le puzzle était chiffré avec la clé publique du client). Le résultat en clair du déchiffrement sera :

  ```
  "S3cr3tT0k3n"
  ```

  qui correspond exactement au token initial attendu.

- **Envoi de la solution** : le client renvoie par exemple `user-3e7a9f-S3cr3tT0k3n` via l'API DO. Le serveur vérifie que le token `"S3cr3tT0k3n"` correspond bien à ce qu'il avait envoyé dans le puzzle, puis autorise la connexion du compte `user-3e7a9f` avec ce mot de passe.

Dans une implémentation réelle, ce processus serait transparent et automatisé via des scripts. L'**interaction typique avec l'API DigitalOcean** consiste en des appels REST sécurisés (HTTPS) pour lire ou modifier les propriétés du droplet comme illustré dans les exemples de commandes `curl` ci-dessus.

Langages et composants utilisés

Le processus d'authentification s'appuie sur plusieurs langages et outils, chacun ayant un rôle spécifique :

- **Shell (Bash)** – Le script principal d'orchestration est écrit en Shell. Il enchaîne les appels (génération de clés, requêtes API DigitalOcean, création d'utilisateurs…) et constitue le cœur logique du processus côté serveur et client.
- **C/C++** – Certaines étapes font appel à du code en C/C++ pour la gestion de la mémoire et des opérations bas niveau. Par exemple, la génération ou la manipulation de données cryptographiques lourdes (calcul RSA, gestion de gros nombres aléatoires) peut être déléguée à des utilitaires ou bibliothèques écrits en C/C++ pour des raisons de performance et de contrôle précis.
- **Lisp** – Un interpréteur Lisp embarqué est utilisé pour certaines logiques métier de haut niveau. Notamment, le projet inclut des scripts Lisp (via SBCL) pour la génération des clés RSA et possiblement pour exprimer la logique du puzzle de manière déclarative. Le Lisp permet d'écrire des composants de logique embarquée (par exemple, dans la génération du nom de droplet ou le calcul de hash pour nommer les clés) avec concision et flexibilité.

Ces différents composants travaillent ensemble. Par exemple, un script Shell peut invoquer un snippet Lisp pour générer une clé (`sandbox_auth.lisp` est exécuté depuis `setup-auth.sh` [3]), ou utiliser un programme utilitaire C pour chiffrer/déchiffrer une charge utile. Le choix de ces langages vise à bénéficier de leurs points forts respectifs dans le cadre de cette authentification sécurisée.

Contraintes de sécurité et remarques

- **Clé publique en clair côté client** : Le fait de stocker la clé publique du serveur en clair sur le client est intentionnel et ne compromet pas la sécurité cryptographique (une clé publique est, par définition, destinée à être connue). Néanmoins, il faut assurer son authenticité (par exemple, en la téléchargeant depuis une source fiable ou en la générant localement via un mécanisme contrôlé) afin d'éviter toute attaque de type « homme du milieu » remplaçant la clé publique par une autre. Dans notre système, la clé publique du serveur peut être exportée de manière automatisée via les scripts d'initialisation, garantissant que Codex utilise la bonne clé [2].
- **Token éphémère et usage unique** : Le token utilisé pour le puzzle est à usage unique et très temporaire. Même s'il transite via le nom du droplet (en clair) à un moment donné, il n'est valable que pour une courte fenêtre et uniquement pour la création de l'accès éphémère. De plus, il est détruit immédiatement après usage côté client et côté serveur, ce qui limite fortement l'exposition en cas de compromission partielle.
- **API DigitalOcean comme canal sécurisé** : L'échange du puzzle et de la solution via l'API DO se fait sur HTTPS avec un token d'API privé. Il est crucial de garder ce token API confidentiel. L'utilisation de l'API évite d'ouvrir un port d'écoute spécifique sur le serveur pour l'authentification, réduisant la surface d'attaque. Néanmoins, cela implique une dépendance envers DO et une légère latence due aux appels REST.
- **Script déclenché par le mot de passe temporaire** : Côté serveur, un script ou mécanisme est mis en place pour **réagir à l'utilisation du mot de passe temporaire**. Par exemple, l'utilisation

d'un mot de passe spécifique pour une connexion peut être interceptée via PAM ou un shell de login personnalisé afin de déclencher la création du compte utilisateur juste-à-temps. Cette approche garantit qu'aucun compte n'existe avant la résolution du puzzle, et que le compte est supprimé juste après son utilisation, renforçant la nature éphémère de l'accès.

- **Nettoyage post-tâche** : Aucune trace durable des éléments secrets ne doit subsister. Les journaux peuvent être épurés pour retirer d'éventuelles occurrences du token (par ex., éviter que le token apparaisse en clair dans un historique de commande ou log d'audit). Le droplet lui-même, s'il est réutilisé pour de multiples tâches, retrouve un état neutre (pas de comptes résiduels, nom réinitialisé, etc.). Si le droplet est dédié à une unique tâche, il pourrait même être détruit complètement en fin de cycle pour une isolation maximale.

En résumé, ce protocole concilie **sécurité** (grâce à la cryptographie asymétrique et l'élimination des accès persistants) et **flexibilité** (utilisation de l'API cloud pour orchestrer l'échange sans interaction manuelle). Il constitue un rituel d'accès contrôlé garantissant que seul un agent ayant fait ses preuves (en résolvant le puzzle) puisse intervenir, et ce uniquement dans les limites imparties de la tâche.

Tâches déléguables à Codex

Voici une liste de tâches pouvant être confiées à Codex (en mode Architecte ou Codeur) pour implémenter ce système étape par étape. Chaque tâche est indépendante autant que possible (certaines dépendent logiquement des précédentes, ce qui est indiqué), ce qui permet de les réaliser en parallèle si nécessaire :

1. **Initialisation de la clé publique et du token**
   Description: Générer la paire de clés RSA serveur et le token secret initial. Exporter la **clé publique** du serveur pour qu'elle soit accessible côté client (Codex) et conserver la clé privée sur le serveur.
   Résultat: Une clé RSA privée stockée (ex: `~/.ssh/id_rsa_auth`) et sa clé publique correspondante partagée (exposée dans une variable d'env or via fichier). Un token secret est également créé en mémoire ou en variable pour l'étape suivante.
   Fichiers/Modifications: Script `auth/setup-auth.sh` (ou équivalent) pour inclure la génération du token en plus de l'appel Lisp existant [3]. Le fichier de clé publique `~/.ssh/id_rsa_auth.pub` est créé/actualisé. *(Dépendances:* aucune, tâche initiale.)*

2. **Publication du puzzle via DigitalOcean**
   Description: Construire le puzzle asymétrique à partir du token (par chiffrement et/ou signature) et publier ce puzzle sur le droplet DigitalOcean via l'API. Cela implique de formater le nom du droplet (ou les tags) pour y insérer le puzzle chiffré, éventuellement avec un préfixe clair. Utiliser le token d'API DO pour effectuer la requête REST.
   Résultat: Le droplet serveur est renommé (ou taggé) avec les informations du puzzle (exemple de nom : `puzzle-<HEX|BASE64_PUZZLE>`). Le puzzle est maintenant accessible au client via l'API DO.
   Fichiers/Modifications: Nouveau script (ex: `auth/publish_puzzle.sh`) contenant l'appel `curl` approprié à l'API DigitalOcean. Ce script utilise les variables `$DO_API_TOKEN`, l'ID du droplet, et le token calculé. *(Dépendances:* Tâche 1 doit avoir fourni le token et la clé publique.)*

3. **Récupération et déchiffrement du puzzle côté client**
   Description: Implémenter dans Codex la récupération du nom (ou métadonnées) du droplet via l'API DigitalOcean, puis extraire le puzzle chiffré. Ensuite, déchiffrer ou vérifier ce puzzle à l'aide

des clés disponibles côté client (par ex., utiliser la clé privée du client si le puzzle a été chiffré pour elle, et la clé publique du serveur pour vérifier toute signature). Assurer la validation du token solution.
Résultat: Le client obtient le **token en clair** après avoir résolu le puzzle. Ce token est stocké dans une variable temporaire (puis sera détruit après usage). La validité du puzzle est confirmée (authenticité vérifiée).
Fichiers/Modifications: Code dans le client Codex (par ex. un module Python ou script Bash) pour appeler l'API (`curl` / `requests`) et pour réaliser le déchiffrement. Possibles modifications dans `codex/auth_client_logic.*` ou création d'un utilitaire `decrypt_puzzle.cpp` si une aide C++ est utilisée pour le déchiffrement. *(Dépendances:* Tâche 2 terminée, accès à la clé privée client et clé publique serveur configurés à l'étape 1.)*

4. **Transmission de la solution et de l'identifiant temporaire**
   Description: Envoyer la solution du puzzle (token) au serveur, accompagnée de l'identifiant utilisateur temporaire qui sera utilisé pour la session. Il s'agit de mettre à jour, via l'API DO, le nom du droplet (ou un tag) en y incluant `username` et `token` solution. S'assurer que le format est conforme à ce que le serveur attend pour la lecture.
   Résultat: Le droplet serveur voit son nom modifié, par ex. `user-<HEX_ID>-<TOKEN>`, attestant de la réussite du challenge. Ces données sont désormais présentes dans les métadonnées du droplet prêtes à être lues côté serveur.
   Fichiers/Modifications: Script ou fonction d'envoi (peut être ajouté à `auth/publish_puzzle.sh` ou un nouveau `auth/send_solution.sh`) pour effectuer le `curl` de mise à jour du nom avec les nouveaux paramètres. *(Dépendances:* Tâche 3 terminée avec un token valide; identifiant temporaire peut être généré soit côté serveur (fourni dans le puzzle) soit calculé ici.)*

5. **Détection et validation côté serveur**
   Description: Développer le composant serveur qui surveille les changements de métadonnées du droplet. Ce peut être une boucle Shell interrogeant régulièrement l'API DO ou un appel déclenché via un webhook. Ce composant doit détecter que le droplet porte désormais l'identifiant et le token solution, puis **valider** que le token correspond au secret initial. En cas de correspondance, lancer la procédure de création de l'accès éphémère.
   Résultat: Le serveur confirme l'authentification réussie. Un indicateur d'état peut être mis à jour (variable, fichier lock) pour signaler que le client est autorisé à se connecter. Si le token ne correspond pas, l'accès est refusé (éventuellement log de l'incident).
   Fichiers/Modifications: Un script Shell (ex: `auth/check_solution.sh`) pouvant être exécuté en tâche de fond sur le serveur. Il utilise `curl` vers l'API DO pour lire le nom du droplet et vérifier le format attendu. Peut également être implémenté en C pour efficacité. *(Dépendances:* Tâche 4 complétée; nécessite DO_API_TOKEN côté serveur aussi ou utilisation de la **Metadata** interne du droplet si disponible).*

6. **Création de l'utilisateur temporaire côté serveur**
   Description: Mettre en place la création automatisée du compte utilisateur éphémère une fois le token validé. Cette tâche inclut la génération d'un nom d'utilisateur hexadécimal (s'il n'a pas été prédéfini) et l'appel aux commandes système pour créer le compte et définir son mot de passe (le token). Intégrer également la clé publique du client dans `~/.ssh/authorized_keys` de ce compte pour autoriser l'accès par clé.
   Résultat: Un nouvel utilisateur (ex: `user-3e7a9f`) existe sur le système, avec mot de passe temporaire configuré et éventuellement clé SSH autorisée. Ce compte est prêt à accepter la connexion de Codex.
   Fichiers/Modifications: Script shell administratif (ex: `auth/create_temp_user.sh`) exécuté sur

le serveur, utilisant des commandes comme `useradd`, `echo 'user:pass' | chpasswd`, et en modifiant `/home/user/.ssh/authorized_keys`. Éventuelle modification de la configuration SSH/PAM pour autoriser la création à la volée. *(Dépendances:* Tâche 5 doit avoir validé le token; certaines informations de Tâche 4 comme le nom d'utilisateur hex sont requises.)*

7. **Test de connexion et exécution de la tâche**
   Description: Vérifier que Codex peut se connecter avec le nouvel utilisateur et exécuter la tâche prévue. Cette étape consiste à initier la connexion SSH (ou autre protocole) en utilisant les crédentials éphémères et à s'assurer que les permissions sont correctes une fois connecté. Codex réalisera l'action prévue (par exemple, lancer un script de déploiement) pour valider toute la chaîne.
   Résultat: Connexion établie avec succès en utilisant `user-3e7a9f` et le mot de passe (ou clé) correspondante. La tâche métier (fictive ou réelle) est effectuée, prouvant que le processus d'authentification fonctionne de bout en bout. Les logs de connexion peuvent montrer l'utilisateur temporaire actif.
   Fichiers/Modifications: Aucune modification de code, mais réalisation d'un **test** de bout en bout. Un script de test (`auth/test_auth_flow.sh`) peut éventuellement automatiser cette connexion (par ex. via `ssh user-3e7a9f@host ...`). *(Dépendances:* Tâche 6 terminée; accès réseau SSH au serveur.)*

8. **Nettoyage post-tâche**
   Description: Automatiser la révocation de l'accès une fois la tâche terminée. Cette tâche supprime le compte utilisateur temporaire du système et nettoie les métadonnées du droplet sur DO. Elle peut être planifiée à la fin du script d'exécution ou déclenchée par Codex une fois sa session terminée.
   Résultat: L'utilisateur temporaire est supprimé (`userdel` + purge du home), son entrée d'authorized_keys nettoyée, et le droplet retrouve un nom neutre (ex: suppression du token dans le nom ou retour à un hostname par défaut via l'API). Le système revient en état sûr, sans trace exploitable de l'authentification passée [6].
   Fichiers/Modifications: Script `auth/cleanup_auth.sh` réalisant les commandes de suppression d'utilisateur et reset des métadonnées DO. Il peut aussi révoquer la clé publique du client de `authorized_keys` si elle y a été ajoutée. *(Dépendances:* Tâche 7 achevée, afin de ne nettoyer qu'après déconnexion.)*

Chaque tâche ci-dessus peut être traitée indépendamment dans la mesure du possible. Par exemple, la récupération du puzzle (Tâche 3) peut être développée et testée en utilisant un puzzle fictif sans avoir besoin que la création d'utilisateur (Tâche 6) soit déjà en place. De même, la création du puzzle (Tâche 2) peut être testée isolément pour vérifier le format de publication sur DO. En cas de dépendance forte, elle est signalée afin de planifier l'**orchestration** dans le bon ordre. Cette décomposition en tâches permet à Codex (mode Architecte/Codeur) d'implémenter le protocole d'authentification de manière modulaire et parallèle, tout en assurant la cohérence de l'ensemble une fois intégré.

---

[1] [4] [5] [6] AGENTS.md
https://github.com/Mikiish/codext/blob/513321706e7563911fdea807c9de2dee7383f572/auth/AGENTS.md

[2] [3] README.md
https://github.com/Mikiish/codext/blob/513321706e7563911fdea807c9de2dee7383f572/auth/README.md