Rapport de Projet (rpc ,rmi ,docker)

***SecurePassManager***

Réalisé par : **Ismail Layachi**

Encadrant : **A.Amamou**

Filière : **Cyber Security**

Année universitaire : **2024 / 2025**

Rapport de Projet : SecurePassManager

Date : 18/06/2025

# 1. Présentation Générale

SecurePassManager est une application Java permettant à un utilisateur de stocker, rechercher, modifier et consulter ses mots de passe personnels de manière sécurisée. Elle repose sur une architecture client-serveur via Java RMI, avec des interfaces graphiques en Swing. Des mécanismes de sécurité comme le hachage, la validation d’entrée et une communication chiffrée via SSL sont intégrés.

# 2. Objectifs Pédagogiques

- Maîtriser les appels de procédures distants (RPC) en Java à travers RMI.  
- Créer des interfaces graphiques ergonomiques avec Java Swing.  
- Mettre en œuvre la sécurité des données : hachage SHA-256, validation, SSL.  
- Comprendre les points faibles des communications non sécurisées.

# 3. Fonctionnalités Implémentées

- Interface d’authentification avec login/register.  
- Gestion complète des mots de passe (ajout, liste, modification, suppression, recherche).  
- Interface serveur avec boutons de contrôle et horloge.  
- Enregistrement sécurisé des mots de passe avec chiffrement.  
- Communication client-serveur sécurisée avec keystore SSL.

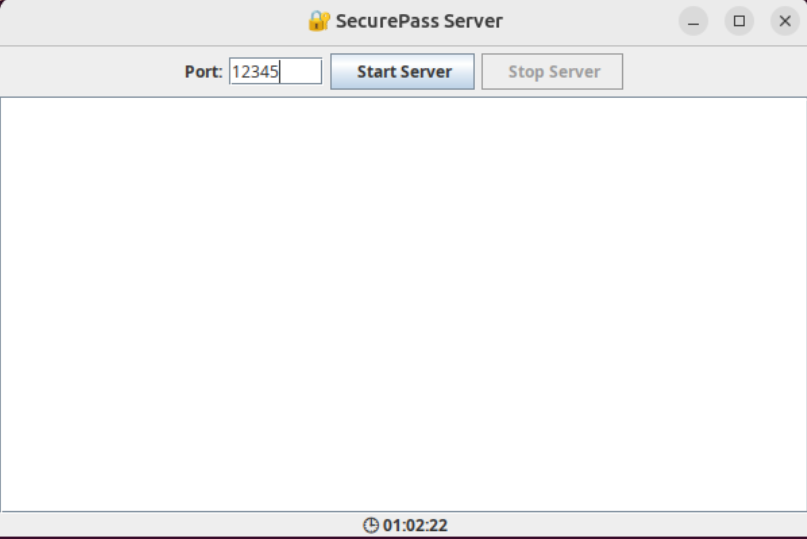


Figure 1 : Interface serveur (avant démarrage)  
Cette interface permet de configurer le port d’écoute du serveur RMI. L’utilisateur peut entrer un numéro de port (par défaut 12345) et cliquer sur 'Start Server' pour initialiser la communication. À ce stade, le serveur est prêt à être lancé.

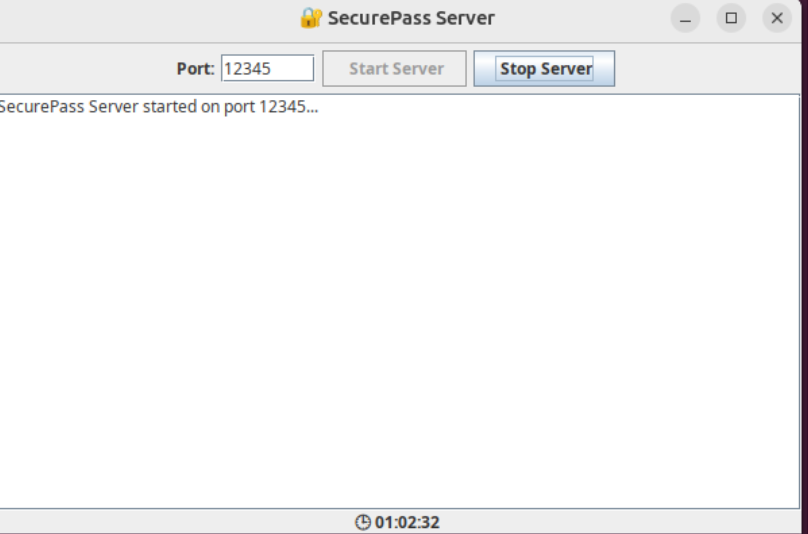


Figure 2 : Interface serveur (après démarrage)  
Une fois le bouton 'Start Server' cliqué, le serveur démarre et affiche un message de confirmation. Le bouton devient inactif et le bouton 'Stop Server' devient accessible.



Figure 3 : Interface client - fenêtre de connexion  
L’utilisateur peut saisir son nom d’utilisateur et mot de passe pour se connecter. Il peut aussi s’enregistrer s’il n’a pas encore de compte.

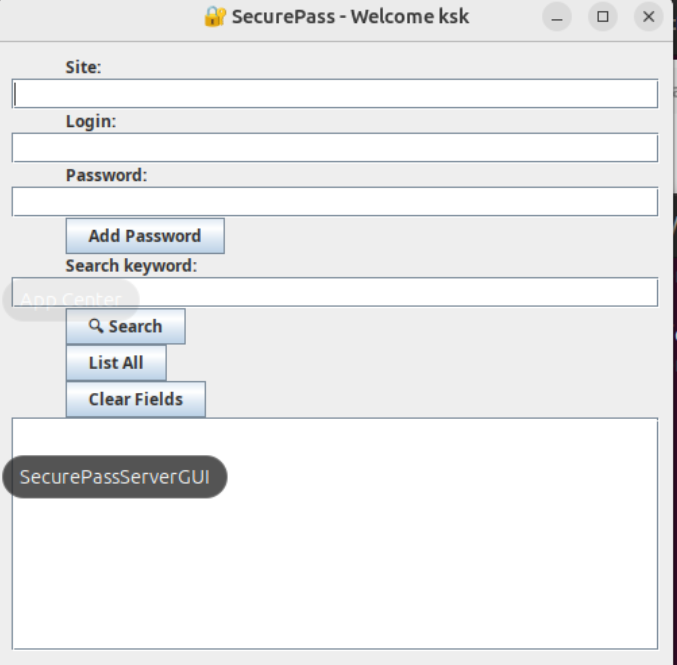


Figure 4 : Interface client - gestion des mots de passe  
Après connexion, l’utilisateur accède à l’espace principal de gestion : ajout de nouveaux mots de passe, recherche par mot-clé, affichage de tous les mots de passe et nettoyage des champs.

# 4. Structure du Projet

📁 common/  
 - VaultEntry.java : modèle de données pour un mot de passe.  
 - VaultService.java : interface RMI déclarant les opérations à distance.  
  
📁 server/  
 - VaultServer.java : classe principale du serveur RMI.  
 - VaultServiceImpl.java : implémentation des fonctions RMI.  
 - SecurePassServerGUI.java : interface Swing côté serveur.  
  
📁 client/  
 - VaultClient.java : interface graphique principale.  
 - TestLoginRegister.java : écran de connexion et d’enregistrement.  
  
📁 util/  
 - CryptoUtils.java : hachage SHA-256.  
 - SecurityUtils.java : outils de validation.  
  
🔐 keystore.jks : certificat pour activer le SSL dans les connexions RMI.

# 5. Aspects Sécurité

- Hachage sécurisé des mots de passe avec SHA-256.  
- Utilisation d’un keystore SSL (`keystore.jks`) pour chiffrer les connexions RMI.  
- Validation des entrées utilisateur pour éviter les injections.  
- Contrôle d’accès avec gestion basique des sessions.

# 6. Déroulement de l'Exécution

1. Démarrer le registre RMI : `rmiregistry`  
2. Lancer le serveur : `VaultServer.java`  
3. Lancer le client : `VaultClient.java`  
4. Se connecter ou s’enregistrer.  
5. Ajouter, modifier, supprimer ou rechercher des mots de passe.

# 7. Problèmes rencontrés et solutions

- Problème de connexion SSL résolu en générant un keystore valide.  
- Problème de synchronisation client-serveur corrigé par l’implémentation de threads.  
- Vérification systématique des entrées pour éviter crashs ou injections.

### 🔒 Projet 2 : SecureVaultRMI – Gestionnaire de Mots de Passe avec Java RMI

Le projet **SecureVaultRMI** est une application Java distribuée développée dans le but de démontrer l’usage de la technologie **Java RMI (Remote Method Invocation)** dans un contexte réel de gestion sécurisée de données sensibles. Il s'agit d'une évolution structurée du premier projet basé sur des sockets, en intégrant un système d’appel de méthodes distantes standardisé, typique des architectures client-serveur Java natives.

#### 🧱 Architecture du système

L’architecture repose sur une séparation claire entre :

* **Le client** : doté d’une interface utilisateur Swing simple et intuitive, qui permet la connexion de l’utilisateur, l’ajout, la recherche et la consultation des mots de passe.
* **Le serveur** : responsable de l’implémentation des services à distance et de l’hébergement des données (en mémoire).

Les composants principaux sont :

* VaultService.java : une **interface distante** (extends Remote) qui définit toutes les opérations disponibles à distance, comme addEntry, getAllEntries, searchEntries, etc.
* VaultServiceImpl.java : la **classe serveur** qui implémente l’interface distante. Elle contient la logique de gestion des mots de passe et d’authentification.
* VaultEntry.java : une **classe sérialisable** représentant un enregistrement de mot de passe (site, identifiant, mot de passe chiffré).
* VaultServer.java : point d’entrée côté serveur qui crée l’objet RMI et l’enregistre dans le registre (rmiregistry).
* VaultClient.java : le **client graphique Swing** qui permet à l’utilisateur final d’interagir avec le serveur via RMI.

#### 🔐 Mécanismes de sécurité

La sécurité est intégrée à plusieurs niveaux :

* 🔑 **Hachage des mots de passe** avec **SHA-256** avant enregistrement, grâce à la classe HashUtil.java. Cela garantit que les mots de passe ne sont jamais stockés en clair.
* ✅ **Validation des entrées utilisateur** pour éviter les injections ou erreurs logiques (champs vides, données invalides).
* 🔒 **Communication sécurisée possible** : bien que le projet utilise RMI classique, il est prévu pour être facilement sécurisé avec un SecurityManager Java et un fichier de type keystore pour le chiffrement SSL.

#### 🎨 Interface utilisateur (Swing)

L’interface utilisateur Swing est ergonomique, divisée en deux étapes principales :

1. **Connexion / Enregistrement** : l’utilisateur entre son nom et mot de passe pour se connecter ou s’enregistrer.
2. **Gestion du coffre-fort** : une fois connecté, il peut :
   * Ajouter un mot de passe pour un site spécifique.
   * Rechercher un mot de passe par mot-clé.
   * Lister tous les mots de passe enregistrés.
   * Effacer les champs ou réinitialiser la vue.

Chaque action du client provoque un appel à distance vers le serveur grâce aux méthodes de VaultService.

#### 🎯 Objectifs pédagogiques atteints

Ce projet vise plusieurs compétences clés :

* **Maîtrise de Java RMI** : compréhension du registre RMI, du mécanisme de liaison (bind), de l’interface distante et des objets sérialisables.
* **Développement distribué** : conception d’une architecture où les traitements sont répartis entre client et serveur.
* **Sécurité des données** : implémentation de bonnes pratiques de hachage et possibilité d'ajouter une couche SSL.
* **Interface graphique** : création d’une application Swing conviviale, complète et fonctionnelle.
* **Sérialisation et RMI** : gestion d’objets transmis à distance entre JVMs différentes via Serializable

## ****Tableau Comparatif – RPC personnalisé vs RMI****

| **Critère** | **SecurePassManager (RPC personnalisé)** | **SecureVaultRMI (Java RMI)** |
| --- | --- | --- |
| **Technologie** | Sockets TCP + objets Request | API Java RMI |
| **Sérialisation** | Manuelle via ObjectInputStream | Automatique avec Remote |
| **Développement** | Plus flexible, plus complexe | Standardisé, plus rapide à coder |
| **Interopérabilité** | Peut être adapté à d'autres langages | Limité à Java |
| **Sécurité** | Personnalisable | SSL possible avec keystore |
| **Performances** | Meilleures si optimisé | Moins performant en surcharge |
| **Installation** | Aucun service requis | Nécessite rmiregistry |
| **Déploiement Docker** | Facile | Plus délicat à exposer |

## 🐳 Partie 3 : Déploiement Docker de SecurePassManager (RPC personnalisé)

Le projet **SecurePassManager**, basé sur une architecture RPC personnalisée via **sockets TCP**, a été conçu pour être facilement déployé dans un environnement conteneurisé à l’aide de **Docker**. L'utilisation de sockets Java permet une grande autonomie du protocole de communication, ce qui rend le projet parfaitement adapté à une architecture conteneurisée ou en microservices.

### 🎯 1. Objectifs de la Dockerisation

* ✅ Faciliter le **déploiement rapide** et reproductible sur n’importe quelle machine.
* ✅ Créer un environnement **isolé et stable** pour exécuter le serveur.
* ✅ Préparer le projet pour un usage **dans le cloud** ou en architecture microservices.
* ✅ Éviter les erreurs liées à la configuration locale (dépendances Java, conflits de ports...).

### 🛠️ 2. Structure Docker

Le projet contient un fichier Dockerfile pour le serveur RPC personnalisé :

dockerfile

CopyEdit

FROM openjdk:17

WORKDIR /app

COPY . .

RUN javac server/SecurePassServer.java

EXPOSE 12345

CMD ["java", "server.SecurePassServer"]

Ce fichier :

* Utilise l'image officielle openjdk.
* Copie tous les fichiers source dans /app.
* Compile le serveur Java.
* Expose le **port 12345** pour la communication socket.
* Lance automatiquement le serveur.

💡 Côté client, tu peux utiliser un simple script run\_client.sh ou lancer l'application localement pour se connecter au serveur conteneurisé.

### 💻 3. Commandes Docker à exécuter

bash

CopyEdit

# Construction de l’image Docker

docker build -t securepass-server .

# Exécution du conteneur serveur

docker run -p 12345:12345 securepass-server

Tu peux ensuite lancer le **client Java Swing localement** pour établir une connexion avec le serveur via localhost:12345.

### ✅ 4. Avantages du Docker pour ce projet

| **Avantage** | **Détail** |
| --- | --- |
| **Portabilité** | Le serveur peut être exécuté sur n'importe quelle machine avec Docker |
| **Isolation** | Pas de conflit de port ou dépendance locale |
| **Reproductibilité** | Même comportement sur toutes les machines |
| **Intégration Cloud** | Facile à déployer dans Kubernetes, AWS, etc. |
| **Travail en équipe** | Chaque membre peut exécuter la même version facilement |

### 🔚 Conclusion

Grâce à cette dockerisation, **SecurePassManager** devient une application prête pour le déploiement en environnement réel ou en infrastructure cloud. Cela renforce son utilité pédagogique en montrant comment un projet Java orienté sécurité peut s’intégrer dans un workflow DevOps moderne.