Rapport du projet Systèmes d'Information
Privacy by Design

2015-2016

Système de récupération de fichiers (Recovery)

Encadrants :

M. Philippe PUCHERAL
M. Luc BOUGANIM
M. Nicolas ANCIAUX
M. Iulan Sandu Popa
M. Paul Tran Van
M. Julien Loudet

Réalisé par :

Soukayna LAFTEH
Oussama BELMEJDOUB
Wassim BENYOUSSEF
Mehdi ZEGAL
Karim ASSAAD

Sommaire

I- Objectif général du projet et mise en situation:

Dans l'ère du numérique, les données sont une ressource essentielle à notre société. On retrouve partout des flux de données sous diverses formes et l'enjeu clé pour des professionnels de l'informatique est d'assurer de la sécurisation de ces données en assurant d'un certain niveau de protection pour dissuader les éventuelles intrusions. L'objectif de notre projet est donc de développer une application d'échange de fichiers sécurisée sur des serveurs personnels qui pourront communiquer entre eux pour s'échanger des fichiers. Ces serveurs seront modélisés par ce que l'on appelle des cellules de confiance, que nous appellerons Token. Dans ces Token sont stockés les fichiers et les données d'une personne, qui peut par la suite connecter son Token à un PC soit pour transférer les fichiers sur ce PC soit pour envoyer des données à un autre Token. C'est dans cette partie que nous intervenons : nous devons nous assurer que la communication entre l'ordinateur et le Token soit sécurisé, ainsi que l'envoi de données à un autre Token. Pour cela, le travail a été divisé en 6 groupes :

- Un premier groupe va s'occuper du contrôle d'accès aux fichiers de la base de données: une interface Homme-Machine permettra de choisir pour chaque fichier quelles sont ses droits d'accès, c'est-à-dire avec qui on peut le partager
- Un autre groupe devra proposer des règles de partage sur les fichiers, par exemple un fichier ne peut être partagé que pendant une période précise, ou d'autres règles sur la localisation. Une API permettra de choisir quelle règle imposer pour chaque fichier.
- Un groupe devra implémenter une interface qui permettra la gestion des fichiers dans le Token à savoir importer, extraire, modifier, renommer ou autre. Les fichiers devront respecter les fonctionnalités implémentées par les deux groupes précédents (contrôle d'accès et d'usage).
- Un autre groupe devra permettre une interaction avec les navigateurs web Firefox ou Thunderbird : il sera possible de télécharger des fichiers directement dans le Token. D'autres interactions tel que l'accès au Token lorsqu'on souhaite ajouter une pièce jointe dans un mail ou autre sont envisagées.
- Un groupe devra implémenter une fonctionnalité qui permettra de stocker dans le Token, si l'utilisateur le veut, les données remplies dans des formulaires sur le net (formulaires classiques tels que nom, prénom, adresse, pseudonyme …) ainsi que les mots de passe si possible. Ainsi, en se connectant avec son Token sur n'importe quel PC, l'utilisateur aura la possibilité d'avoir ses formulaires déjà remplis.
- Enfin, notre groupe aura comme tâche d'implémenter un serveur de secours, qui stockera toutes les données des Tokens pour que les utilisateurs puissent les récupérer en cas de perte de leur Token. Nous vous expliquerons ce travail plus précisément dans la prochaine partie.

A noter que toutes ses fonctionnalités doivent respecter des règles bien précises en terme de sécurité, à savoir que les fichiers contenus dans les Tokens doivent être chiffrés à l'aide de l'algorithme de chiffrement AES « Advanced Encryption Standard » un algorithme de chiffrement symétrique donc la même clé permet de chiffrer et déchiffrer.

II- Objectifs fonctionnels du système de récupération:

Notre partie consiste ainsi à stocker dans un serveur de secours toutes les données de toutes les TCells des différents utilisateurs. Ce serveur de secours sera modélisé par une TCell spécifique, avec une base de données **SQLite** structurée différemment. Les fichiers stockés dans ce serveur doivent ensuite pouvoir être récupérable par leur propriétaire d'origine au cas où il perd certains fichiers ou s'il perd sa TCell. Nous allons dans un premier temps expliquer le fonctionnement de la première phase qui est la sauvegarde des données dans le serveur de secours, puis nous verrons comment les propriétaires peuvent récupérer leurs données de deux manières différentes.

Pour recevoir toutes les données d'une TCell, le serveur de secours devra intercepter toutes les commandes qui entrainent la réception d'un fichier dans la TCell. Le serveur devra ainsi intercepter la commande « **StoreFile** », qui entraine un transfert de données d'une application vers la TCell, ainsi que la commande « **ShareFile** » qui indiquera la réception d'un fichier envoyé par une autre TCell. Nous ajouterons ensuite dans le code de ces commandes l'appel d'autres commandes qui permettront d'envoyer le fichier au serveur de secours. Du côté serveur, un daemon qui tourne en permanence attend une commande, dès qu'il en reçoit une, suite à un « ShareFile »ou un « StoreFile » il s'occupe de stocker correctement le fichier dans la base de données en ajoutant un identifiant spécifique à chaque fichier. Nous vous détaillerons plus spécifiquement cette partie dans la description technique de la base de données SQLite du serveur.

Pour la récupération de données, un utilisateur devra utiliser la commande « **SendRecoverRequestFile** » ou « **SendRecoverRequestShamir** » selon le choix de l'utilisateur. Dans un premier temps, nous avons fait en sorte que l'utilisateur devait se rappeler de sa clé privée pour pouvoir récupérer ses fichiers. Ensuite, pour avoir plus de sécurité, nous avons utilisé le principe du partage de clé secrète de Shamir. L'idée est que pour pouvoir avoir accès à ses fichiers, l'utilisateur partage sa clé privée en n parties, il donne chacune de ces parties à l'un de ces amis. Cependant, l'avantage du secret de Shamir est que pour déchiffrer les fichiers du serveur de secours et donc récupérer ses fichiers intégralement, l'utilisateur n'a pas besoin de réunir ces n parties, il lui faut simplement qu'un nombre k<n choisi de ces amis lui donnent leur partie qu'il leur a confié pour pouvoir faire la récupération.

III- Description technique du système de récupération:

1- Structure du Recovery Server :

Le Recovery Server a été fait avec une base de données SQLite comportant **4 tables** (toutes les tables comportent un Id qui s'incrémente de façon automatique) :

- **FILE** : pour stocker les fichiers reçus par une TCell
  - **UserID** : l'identifiant de l'émetteur
  - **GID** : le UserId concaténé avec le nom du fichier reçu (UserID|FileName)
  - **SKey** : la clé secrète chiffré avec la clé publique du UserID
  - **IV** : le vecteur d'initialisation chiffré avec la clé publique du UserID
  - **Command** : l'origine de la commande qui a émis le fichier

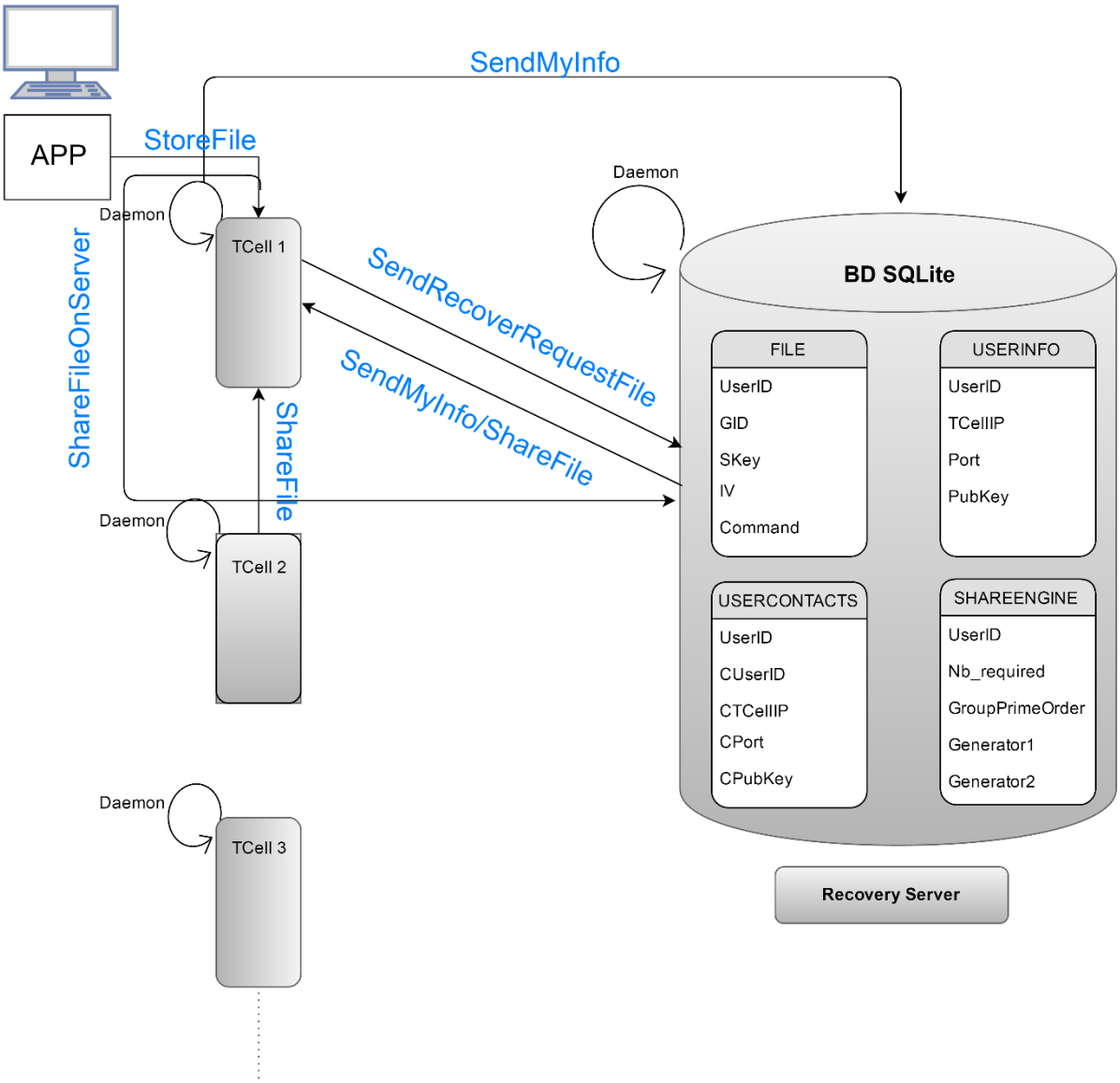
- **USERINFO** : pour stocker les informations personnelles de chaque utilisateur
  - **UserID** : l'identifiant de l'émetteur
  - **TCellIP** : l'IP de l'émetteur
  - **Port** : le port de l'émetteur
  - **PubKey** : la clé publique de l'émetteur

- **USERCONTACTS** : pour stocker les contacts de chaque utilisateur
  - **UserID** : l'identifiant de l'émetteur possédant le contact
  - **CUserID** : l'identifiant chiffré du contact
  - **CTCellIP** : l'IP chiffré du contact
  - **CPort** : le port chiffré du contact
  - **CPubKey** : la clé publique chiffrée du contact

- **SHAREENGINE** : le moteur de Shamir permettant de reconstituer le secret
  - **UserID** : l'identifiant du possesseur du secret
  - **Nb_available** : attribut publique du moteur (le nombre de partage)
  - **Nb_required** : attribut publique du moteur (le nombre requis pour reconstituer)
  - **GroupPrimeOrder** : attribut publique du moteur
  - **Generator1** : attribut publique du moteur
  - **Generator2** : attribut publique du moteur

2- Les commandes ajoutées au système de récupération :

1-1- Version minimale :



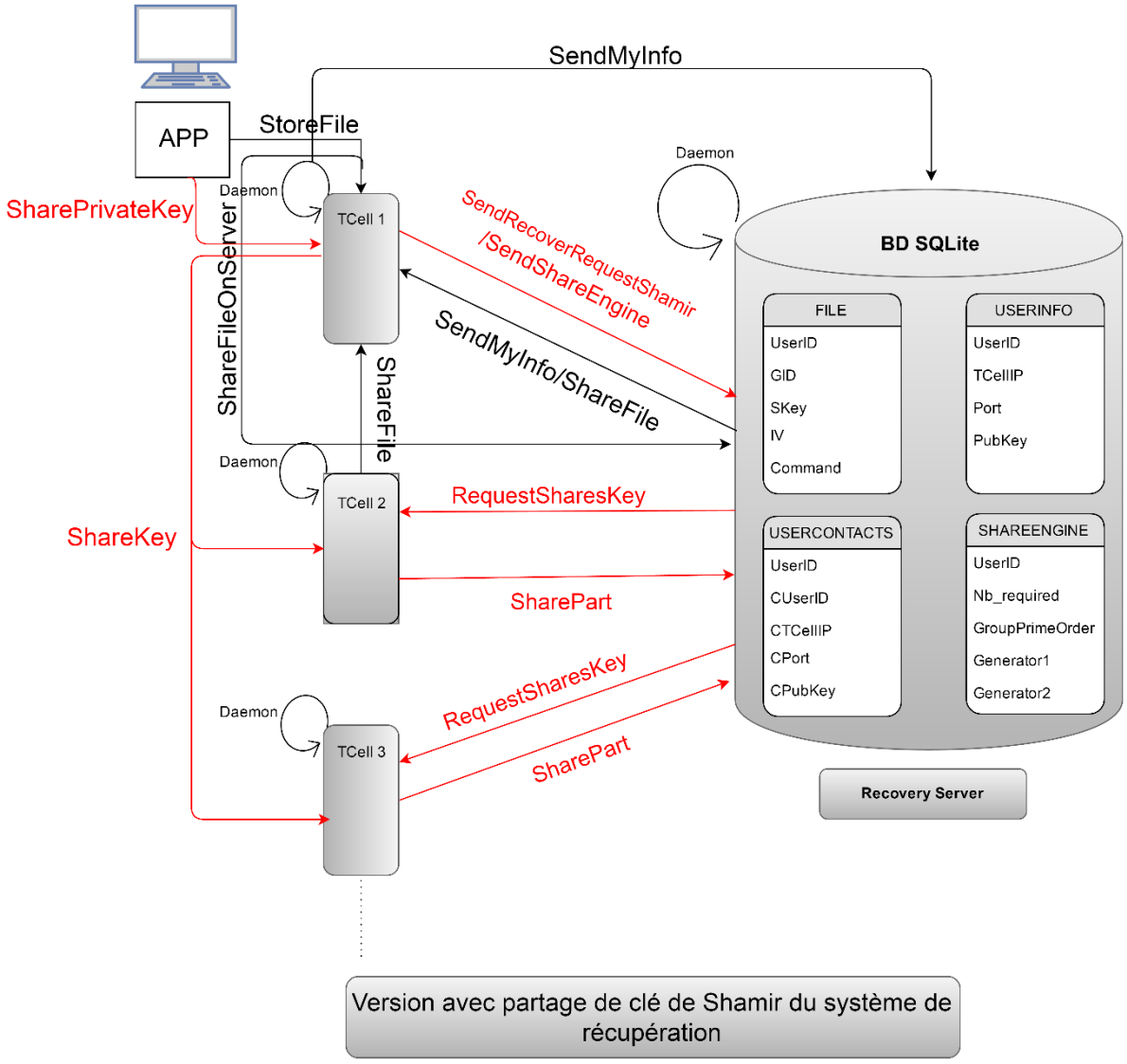
Dans la version minimale, on a supposé que l'utilisateur détient sa clé privée au moment où il demande le recovery. Et pour permettre la communication entre le RecoveryServer et la TCell, il a fallu ajouter les commandes ci- dessous :

1- **ShareFileOnServer** : Elle est appelée dans deux endroits différents qui caractérisent la réception d'un fichier. Au moment où on fait un **StoreFile** et au moment où la Tcell reçoit un fichier à travers un **ShareFile**. Le Recovery Server reçoit donc toutes les informations nécessaires pour pouvoir remplir la table **FILE**.

2- **SendMyInfo** : Elle est appelée une fois le daemon de la TCell est lancé. Elle permet d'envoyer au Recovery Server le contenu de la table **MYINFO** (sans pour autant envoyer la clé privée), et le contenu chiffré avec la clé publique de la table **USER** pour garder la confidentialité de ses contacts. Le contenu de la table MYINFO n'a pas été chiffré parce que le RS a besoin de ces informations pour pouvoir lui renvoyer le recovery.
   La même commande est utilisée au niveau du RS pour permettre l'envoi des informations à la TCell demandant le recovery, ce qui permettra de remplir les deux tables MYINFO et USER.
3- **SendRecoverRequestFile** : Elle consiste la fonction principale de **l'API du client RecoverFromFile**. Elle prend en entrée l'emplacement de la clé privée de l'utilisateur qui demande le recovery. Le RS n'a besoin donc que de renvoyer les informations contenues dans les tables SQLite à travers la commande **SendMyInfo** comme expliqué avant pour les deux tables MYINFO et USER, et aussi à travers la commande **ShareFile** pour la table FILE.

Pour la version minimale, ces commandes sont suffisantes pour assurer le bon fonctionnement du Recovery Server.

1-2 Version avec partage de clé secrète de Shamir



La 2ème version qui intègre le **partage de clé de Shamir** a donc besoin d'autres commandes qu'on explique ci-dessous :

1- **SharePrivKey** : Cette commande consiste la fonction principale de **l'API du client SharePrivKey.** Elle a comme entrée un tableau qui rassemble les IDs des TCells connectées que le client choisit pour leur envoyer chacun une partie de sa clé privée. Elle applique donc à cette clé privée la fonction **shamirShare** qui permet de la diviser sur le nombre choisi. L'envoi à chaque TCell se fait à travers la commande **ShareKey** qu'on explique juste après. Elle appelle aussi la commande **SendShareEngine** qu'on explique aussi tout juste après.

2- **ShareKey** : Elle est appelée au sein de la commande SharePrivKey. Elle s'occupe d'envoyer à la TCell concernée la partie de la clé privée générée par la commande précédente.

3- **SendShareEngine** : Elle est appelée au sein de la commande SharePrivKey. Elle s'occupe d'envoyer le **moteur** généré par l'instanciation de la classe **ShamirShare**. Ce moteur est initialisé par le nombre de partage à effectuer et le nombre de parties requis (déclaré comme une constante) pour permettre la restitution du secret. Ce moteur s'occupe donc de faire le partage du secret et est donc requis pour faire la reconstitution. Elle est donc utilisée aussi par le RS pour renvoyer le moteur à l'utilisateur qui demande le recovery.
4- **SendRecoverRequestShamir** : Cette commande fait le même travail que SendRecoverRequestFile dans la version minimale. La seule différence c'est qu'elle consiste la fonction principale de **l'API du client RecoverFromShamir**. Elle prend en entrée un tableau contenant les IDs des TCells (connectées) qu'on va interroger (voir commande ci-après) pour récupérer chaque partie de la clé. Ces parties vont être ensuite assemblées grâce au moteur déjà stocké dans la BD du RS et qui va nous permettre d'appeler la fonction **shamirRecover** pour reconstruire la clé avant de l'envoyer à l'utilisateur.
5- **RequestSharedKey** : Cette commande est appelée au niveau du RS au moment où il reçoit un SendRecoverRequestShamir. Elle consiste à interroger les TCells choisis par le client pour récupérer la partie du secret qu'elle possède.

3- Compléments techniques :

Pour le **chiffrement/déchiffrement**, on a gardé les mêmes algorithmes utilisés dans la TCell, **AES/CBC** pour le chiffrement du fichier et **RSA** pour le chiffrement de la clé secrète.

La réalisation du **partage de clé de Shamir** se repose sur la base d'une librairie JAR qu'on a pu récupérer sur le net et qu'on a su adapter le code source contenu à notre projet. La source se trouve sur ce site http://shamir-secret-sharing-in-java.soft112.com/.

IV- Description du code du système de récupération:

1- Codes modifiés dans la TCell :

- **Apps.AppMain** : Ajout des **6 exemples** types pour simuler le fonctionnement de notre système de recovery. La variable **test** a été créé pour faciliter le choix de l'exemple voulu. Voici ce que fait AppMain selon chaque valeur donné à test :
  - **0** : Un StoreFile standard pour vérifier la réception du RS du fichier stocké
  - **1** : Un ShareFile standard à deux TCells pour vérifier la réception du RS aux fichiers stockés
  - **2** : Un RecoverFromFile en fournissant le chemin de la clé privée et après avoir vidé la BD de l'utilisateur l'exécutant
  - **3** : Un SharePrivKey pour partager sa clé privée sur le tableau d'utilisateurs entérés (dans notre cas, on a mis 3)
  - **4** : Un RecoverFromShamir en fournissant un tableau d'une taille égale ou supérieure à la constante définie pour le nombre minimal d'utilisateurs requis pour recréer le secret.
  - **5** : La même que le 4 mais avec un tableau différent.
- **Api.ClientAPI** : Ajout de deux attributs à ShareFile.shareFile. Constants.FROM_TCELL, qui permet d'exécuter dans le daemon de la TCell le traitement d'un ShareFile usuel, et la chaîne vide qui correspond à la clé privée qu'on n'aura pas besoin ici (cela aura un sens quand ça sera le RS qui exécutera cette fonction)
- **Command.ShareFileCommand** : Ajout de deux attributs. Source pour dire l'appel est fait à partir d'une TCell ou d'un RS. Et privateKey qui comportera la clé privée de l'utilisateur. Elle ne sera concrètement utilisée que si c'est le RS qui l'exécute.
- **Message.ShareFile** : Ajout de Source et privateKey comme expliqué avant.
- **Message.StoreFile** : A la fin du traitement standard d'un StoreFile, on ajoute une partie qui s'occupe d'envoyer les mêmes informations stockées au RS aussi. La SKey et l'IV sont envoyés de manière chiffrés pour respecter la confidentialité.
- **Tools.Constants** : Pour les 8 commandes définies dans la section III.2, on a ajouté 8 constantes. En plus des 2 constantes, **FROM_TCELL** et **FROM_RS** pour le ShareFile. Et aussi, une constante pour déterminer le nombre minimal requis pour recréer un secret de Shamir. Et finalement, 3 autres constantes décrivant l'état du statut reçu.
- **Tools.Tools** : Ajout des interprétations des nouvelles constantes ajoutées à Constants.
- **Daemon.ClientConnectionManager** :
  - Dans le cas où la TCell intercepte la commande **ShareFile**, si ça vient d'une TCell, on envoie les informations reçus au RS aussi tout en prenant soin de chiffrer la SKey et l'IV avant l'envoi. Sinon si ça vient du RS (demande de recovery), on fait un traitement sur le GID récupéré du serveur pour enlever l'id qu'on a mis au début du nom de fichier et le concaténer au répertoire de la TCell où il faut mettre les fichiers récupérés. Dans le cas où c'est le RS encore une fois, on déchiffre la SKey et l'IV avant d'enregistrer les données sur la TCell.
  - Si la commande **SharePrivKey** est interceptée, on lance la fonction shamirShare sur la clé privée pour la diviser en parties et on envoie à chaque utilisateur choisi

une partie avec la commande ShareKey. Enfin, on envoie le moteur généré au RS pour pouvoir inverser le processus après.
    - Si la commande **ShareKey** est interceptée, cela veut dire que l'utilisateur a reçu une partie d'un secret de son ami. Et donc, il insère cette partie dans la table SharedKey.
    - Si c'est **RequestSharedKey** qui est interceptée, cela veut dire que le RS a demandé une partie d'un secret qu'un utilisateur garde pour un autre utilisateur. Il la cherche dans sa table SharedKey et l'envoie au RS.
    - Et finalement, si c'est **SendMyInfo** qui est interceptée, cela veut dire que le RS envoie à l'utilisateur les informations concernant les 2 tables MYINFO et USER. Il déchiffre le contenu de la table USER avant de l'inséerer dans la BD de la TCell.
- **Daemon.TCellDaemon** : Avant de lancer le ClientConnectionManager, on utilise la commande sendMyInfo pour envoyer le contenu des 2 tables MYINFO et USER. La première est stockée en plaintext dans le RS parce qu'on aura besoin de ces informations pour lancer la récupération. La 2ème est stockée chiffrée en RSA pour garder les informations concernant les utilisateurs qu'ils possèdent anonymes. Pour effectuer ce chiffrement, on a utilisé **EncrUser** plutôt que User pour nous faciliter la tâche de l'envoi et de la réception puisque EncrUser ne définit que des String alors que User définit des entiers qui ne sont pas fourni au moment du chiffrement.
- **Dao** : Le fait que le QGen est changé a fait que tout le dao a changé (voir modifications QGen ci-dessous)
- **Qgen/schema.sql** : Dans la BD de la TCell, on a ajouté la table **SharedKey** qui permet de stocker une partie du secret de Shamir au cas où un utilisateur a décidé de partager sa clé privée via l'API SharePrivKey. Voici la description des attributs de cette table :
    - **IdGlobal** : id auto incrémenté
    - **IND** : l'index de la partie de la clé reçu
    - **ENCRYPTEDSHARE** : attribut de la classe shamir.Share pour identifier la partie reçu
    - **SHARE** : attribut de la classe shamir.Share pour identifier la partie reçu
    - **PROOFC** : attribut de la classe shamir.Share pour identifier la partie reçu
    - **PROOFR** : attribut de la classe shamir.Share pour identifier la partie reçu
    - **U** : attribut de la classe shamir.Share pour identifier la partie reçu
- **Qgen/queries.sql** : l'ajout des requêtes : **insertSharedKey** et **getSharedKey** pour la table SharedKey. Et **deleteMyInfo**, pour pouvoir restocker les informations correctement après avoir demandé le recovery des informations d'un utilisateur. On a choisi cette alternative après avoir remarqué que les requêtes UPDATE ne marchaient pas sur la TCell.

2- Codes ajoutés dans la TCell :

- **Api.ClientAPI** : Ajout de trois APIs : RecoverFromFile, RecoverFromShamir et SharePrivKey.
- **Beans.EncrUser** : Similaire à User mais ne contenant que des String pour permettre l'envoi de données chiffrées d'un utilisateur au RS au moment où il lande son daemon.
- **Command.RequestSharedKeyCommand** : contient le userID de l'utilisateur qui a besoin de récupérer sa partie de la clé.
- **Command.SendMyInfoCommand** : contient un objet MyInfo contenant les inforamtions de la table MYINFO et une liste de EncrUser contenant les informations chiffrées de la table USER.
- **Command.SendRecoverRequestFromFileCommand** : contient le chemin de la clé privée ainsi que l'id de l'utilisateur qui demande le recovery.
- **Command.SendRecoverRequestFromShamirCommand** : contient un tableau de contacts desquels on va reconstituer la clé privée de l'utilisateur userID demandeur du recovery.
- **Command.SendShareEngineCommand** : contient un objet PVSSEngine, qui caractérise le moteur de partage de clé de Shamir, appartenant à l'utilisateur userID.
- **Command.ShareFileOnserverCommand** : contient le fichier à envoyer, le gid (userid|filename), la skey (chiffrée), l'iv (chiffré), le userID et command pour le type de la commande reçu.
- **Command.ShareKeyCommand** : Contient le userID du détenteur de la partie part (de type Share) du secret
- **Command.SendShareEngineCommand** : contient un tableau de contacts pour lesquels on va envoyer à chacun une partie de la clé privée.
- **Messages** : ajout des fonctions statiques à appeler pour envoyer une commande. Pour chaque commande citée au-dessus, on associe un message qui va s'occuper d'ouvrir un **socket** avec soi-même, un utilisateur ou le RS selon la nature de la commande. Et on envoie la commande associée avec un **writeObject** sans oublier de la **sérialiser** pour permettre l'envoi de cet objet. Et après, on attend la réception d'un entier qui sera interprété par la fonction **interpretStatus** du package Tools pour afficher le message correspondant.
- **Shamir** : package contenant le code implémentant le partage de secret de Shamir. Il contient :
    - **InvalidVSSScheme** : exception personnalisée pour le code.
    - **PublicInfo** : les informations requis pour pouvoir reconstituer le secret.
    - **PublishedShares** : contenant les parties du secret partagé.
    - **PVSSEngine** : le moteur qui s'occupe de diviser ou d'assembler des parties d'un secret.
    - **ShamirShare** : classe qu'on a déclarée contenant le moteur et centralisant le traitement en 2 fonctions types, shamirShare et shamirRecover.
    - **Share** : contient les informations d'une partie du secret.

- **Database.DatabaseRecover** : Fait la même chose que DatabaseMain mais laisse les tables MYINFO et USER **vides** sauf le userGID et le port de la table MYINFO qui sont nécessaires pour pouvoir ouvrir une connexion avec le daemon. Elle est principalement utilisée pour simuler la perte de données d'une TCell.

3- Codes du Recovery Server :

Pour le Recovery Server, on est parti du principe qu'il est similaire à une grosse TCell mais qui travaille avec SQLite pour la gestion de la BD. Cela veut dire qu'il contient des éléments similaires au code de la TCell. Dans ce qui suit, on mettra « **similaire à la TCell** » pour tous les éléments qui le sont :

- **Beans** : similaire à la TCell
- **Command** : similaire à la TCell mais sans les éléments suivants : GetAllFilesDescCommand, GetUserCommand, ReadFileCommand, StoreFileCommand et SharePrivKeyCommand.
- **CryptoTools** : contient seulement le KeyGenrator et le KeyManager du même package dans la TCell.
- **Daemon.ClientConnectionManager** : il intercèpte les commandes suivantes :
    - **SendMyInfo** : Pour pouvoir récupérer les données envoyées par la TCell concernant les **2 tables MYINFO et USER** et qu'il stocke respectivement dans les tables USERINFO et USERCONTACTS.
    - **ShareFileOnServer** : Cela veut dire que le RS a reçu des données à travers un STORE ou un SHARE. Donc, dans un premier temps, il crée le fichier reçu dans le chemin par défaut du RS défini comme une constante dans Constants. Et dans un deuxième temps, il insère les données dans la table **FILE** du RS.
    - **GetFileToShare** : Similaire à la TCell (pour pouvoir récupérer le fichier à envoyer au moment où ShareFile est exécutée)
    - **SendRecoverRequestFromFile** : Il génère la clé privée à partir du chemin de la clé privée donné en argument et il exécute un ShareFile vers le demandeur du recovery mais maintenant en spécifiant comme **4ème argument** Constants.FROM_RS pour que le daemon de la TCell sache qu'il faut faire le traitement concernant le RS (on ne va quand même pas renvoyer au RS ce qu'on vient de recevoir du RS) et comme **5ème argument** la clé privée générée pour que l'utilisateur puisse déchiffrer certaines données de sa table FILE. Après, on exécute un SendMyInfo pour envoyer aussi les informations de sa table MYINFO (contenant la clé privée) et de sa table USER qu'il va déchiffrer une fois reçu.

  - **SendRecoverRequestFromShamir** : Elle fait le même travail que la commande juste avant sauf que la génération de la clé privée se fait après avoir lancé la commande **RequestShareKey** sur chaque utilisateur concerné (ceux choisi). Comme ça, le RS récupérera des parties suffisantes pour pouvoir recréer la clé privée. Cette dernière action se fait après que le RS récupère depuis sa BD SQLite à partir de la table **SHAREENGINE** le moteur correspondant à l'utilisateur demandant le recovery. Ce moteur va pouvoir lancer la fonction shamirRecover qui reconstruira le secret.
  - **SendShareEngine** : Intercepter cette commande veut dire qu'un utilisateur a appliqué le partage de clé de Shamir sur sa clé privée et qu'il vient de générer un moteur qui sera primordial pour la reconstruction de son secret. Donc le RS prend ce moteur et l'insère dans la table **SHAREENGINE** pour cet utilisateur-là.
- **Daemon.TCellDaemon** : Similaire à la TCell mais sans l'envoi des informations concernant les 2 tables USER et MYINFO.
- **Dao.TCelDAO** : Contient la déclaration de la BD SQLite comme décrit dans la partie III.1. Elle contient aussi des fonctions exécutant un count sur chaque table pour savoir où il faut insérer la prochaine ligne à chaque fois qu'on relance le daemon. Elle contient des fonctions **d'insertion** dans chacune des trois tables, des fonctions de **récupération** de données (Select), des fonctions **d'affichage** des données, quelques fonctions retournant un booléen par exemple pour savoir si un utilisateur ou un fichier existe déjà dans la table et enfin quelques fonctions de suppression de données selon le besoin.
- **Database.DatabaseMain** : Lance **DropTables** ou vider la BD et ensuite **CreateTables** pour créer les tables.
- **Messages** : Dans le RS, on n'a que **4 messages** définis dont 3 similaires à ceux qui existent dans la TCell qui sont GetFile, SendMyInfo et ShareFile. **RequestSharedKey** qui est propre au RS s'occupe d'envoyer un socket à chaque utilisateur apprtenant au tableau entré tout en levant l'exception « utilisateur inexistant » si la connexion échoue. Chaque partie récupérée est ajoutée au Vector<Share> passé en argument via ce message.
- **Shamir** : similaire à la TCell
- **Tools.Constants** : similaire à celui de la TCell mais sans les constantes de commandes que le RS ne définit pas. On ajoute aussi la constante **RS_PATH** où on stockera les fichiers reçus par les TCells. Et aussi, la constante d'état **SHARED_ENGINE** pour confirmer la bonne réception du moteur de partage de Shamir.
- **Tools.Tools** : Similaire à la TCell avec en plus la fonction **interpretStatusServer** pout interpréter les messages émis par le RS.

V- Répartition du travail:

Après avoir fait le cahier des charges, on a décidé de répartir les tâches pour pouvoir avancer plus vite. Donc Soukayna LAFTEH s'est informé sur SQLite, Karim ASSAAD et Mehdi ZEGHAL sur le chiffrement symétrique et asymétrique et Oussama BELMEJDOUB et Wassim BENYOUSSEF ont essayé de refaire le TP.

Après que chacun ait fini sa tâche, chacun de nous a expliqué pour les autres le travail qu'il a réalisé et ce qu'il vient d'apprendre.

La prochaine étape était maintenant de comprendre le code. La répartition des tâches n'étant pas une bonne idée à cause de la grande dépendance entre les éléments du code et entre chaque package et package et aussi le code étant difficile à comprendre. Nous avons décidé de se réunir pour comprendre le code et pout faire petit à petit l'architecture du Recovery Server, tout en modifiant en même temps et chacun sur sa VM les éléments de la TCell quand on en a besoin, jusqu'à ce que le projet ait abouti.

VI- Conclusion:

Ce projet nous a permis de mettre en pratique les connaissances acquises au cours du module SIPD pour tout ce qui approche Privacy by Design et cryptographie. Il nous a aussi permis de s'entraîner encore plus sur la programmation orientée objet en JAVA.

Le projet nous a appris à chercher des solutions pour parvenir aux problèmes croisés et nous a permis en tant que groupe en discutant ces problèmes de nous projeter vers des situations réelles dans la vie d'entreprise.

Le vrai défi que nous avons eu tous était de comprendre un code contenant plusieurs dépendances entre d'autres parties du code et en créer un similaire et avec encore plus de dépendances.

Nous tenons à remercier tous nos encadrants qui ont su nous guider jusqu'à la réalisation de ce projet.