**Question 1**

La première tâche de ce projet consiste à élaborer un diagramme entité-relation (ER) pour identifier les différentes entités et les relations qui existent entre elles. Ce diagramme nous permet de visualiser les interactions entre des éléments essentiels comme les agents, les incidents, les attaquants, les cibles et les réponses. Par exemple, un agent est associé à plusieurs incidents, tandis qu'un incident peut concerner plusieurs attaquants. Une fois ce modèle ER conçu, nous le traduisons en un modèle relationnel sous forme de tables dans une base de données relationnelle.

Chaque table doit contenir une clé primaire pour garantir l’unicité des enregistrements (par exemple, id\_agent pour les agents). Les clés étrangères assurent la cohérence des relations entre les tables (comme un champ id\_attack dans une table de victimes pour lier chaque victime à une attaque spécifique). Nous adoptons la troisième forme normale (3NF), une norme de structuration qui permet d’éviter les redondances et de minimiser les anomalies lors des insertions, suppressions ou mises à jour.

**Diagramme entité-relation**

Une image contenant texte, écriture manuscrite, diagramme, Plan

Description générée automatiquement

**Modèle physique**Une image contenant texte, écriture manuscrite, mots croisés, nombre

Description générée automatiquement

**Question 2**

Cette question a pour objectif de restructurer les données pour rendre l’exploitation plus efficace. La colonne "Response" du fichier CSV contient des informations regroupant à la fois le type de réponse et la source de cette réponse. Afin de mieux organiser ces informations et faciliter les requêtes, il est nécessaire de séparer cette colonne en deux : une colonne dédiée au type et une autre à la source de la réponse. Cela permet par exemple d'analyser quelles sont les réponses fréquentes ou d'identifier les acteurs les plus actifs dans la gestion des incidents. Cette opération se fait par un traitement du fichier CSV qui, après modification, est sauvegardé sous une nouvelle version contenant ces deux nouvelles colonnes.

**Question 3**

Dans cette étape, nous analysons la colonne "Affiliations" du CSV afin de détecter si certains groupes d’attaquants sont mentionnés comme suspects sans être confirmés. L’idée est de différencier les responsables avérés des suspects, ce qui permet à la DGSE de mieux prioriser ses enquêtes. Pour ce faire, nous ajoutons une nouvelle colonne booléenne à la base de données. Cette colonne indique si le groupe d'attaquants est confirmé (1) ou non (0). En automatisant cette distinction, nous facilitons l’analyse stratégique des incidents.

**Question 4**

L’objectif de cette étape est de charger automatiquement la configuration de la base de données à partir d’un fichier de configuration externe. Le fichier, situé dans le dossier ./config/config/, contient des informations importantes comme le nom de la base de données, son emplacement, et d’éventuelles options de connexion.

Cette automatisation est essentielle pour simplifier le déploiement et éviter des erreurs de configuration manuelle. Plutôt que de modifier le code à chaque changement de paramètres, on peut ajuster directement le fichier de configuration. Le script load\_config() lit ce fichier et transmet les paramètres nécessaires au reste du programme. En cas d'erreur (par exemple, si le fichier est absent ou mal écrit), le système doit afficher un message clair afin que l'utilisateur puisse corriger le problème rapidement.

**Question 5**

Cette étape consiste à créer la base de données à partir du modèle relationnel précédemment défini. À ce stade, toutes les entités (agents, incidents, attaquants, etc.) sont traduites en tables relationnelles. Chaque table comporte ses propres colonnes avec des types de données bien définis :

INTEGER pour les identifiants uniques et les champs booléens (0 ou 1),

TEXT pour les autres paramètres tel que les noms.

La fonction create\_database() crée ces tables et établit les relations entre elles via des clés primaires et étrangères, garantissant ainsi l'intégrité des données. Une fois la base créée, elle est prête à recevoir des données et à être interrogée via SQL.

**Question 6**

L’objectif ici est de peupler la base de données avec des informations extraites du fichier CSV modifié. Le processus de remplissage est essentiel car il permet de rendre immédiatement utilisables toutes les tables que nous avons créées.

Lorsqu’on insère les données, il est primordial que chaque entrée respecte les contraintes relationnelles. Par exemple, chaque incident doit être associé à un agent valide à travers une clé étrangère. Si un incident mentionne un agent qui n’existe pas encore dans la table des agents, cela entraîne une erreur d'intégrité et l’insertion échoue. C’est pourquoi il est nécessaire d’exécuter l’insertion dans un ordre logique : d'abord les agents, puis les incidents, et enfin les relations entre ces entités.

**Question 7**

Cette étape vise à rendre automatique l’initialisation complète de la base de données. La fonction init\_database() est modifiée pour intégrer à la fois la création des tables et leur remplissage immédiat. Cela permet d’éviter des manipulations manuelles répétées chaque fois que l’application est lancée dans un nouvel environnement.

En automatisant ce processus, on s’assure que la base est toujours dans un état cohérent et prête à être utilisée. Une telle approche est utile en production, car elle réduit le risque d’erreur humaine et accélère le déploiement du système.

**Question 8**

Les requêtes paramétrées sont une bonne pratique de programmation pour exécuter des requêtes SQL de manière sécurisée. Dans la fonction insert\_agent(), l’utilisation d’une requête paramétrée permet d’éviter les injections SQL, une des attaques les plus fréquentes sur des bases de données.

Une autre fonctionnalité clé abordée ici est la gestion des erreurs d’intégrité. Par exemple, si on tente d’ajouter un agent avec un identifiant ou un nom déjà existant, SQLite renvoie une IntegrityError. Une gestion appropriée de ces erreurs permet de maintenir la stabilité de l’application et d’informer l’utilisateur de manière claire lorsqu’une insertion échoue.

**Question 9**

Cette question nous amène à identifier pourquoi un agent ajouté n'apparaît pas dans la base. Par exemple, en ajoutant Hubert via une requête, nous constatons qu’il ne figure pas dans la liste des agents. Ce type de problème peut provenir d’une contrainte d'intégrité non respectée ou d’une erreur dans le code de la requête SQL.

Une analyse approfondie du code de création de la table des agents permet de comprendre l’origine de ce problème. Il s’agit d’un oubli de valider la requête avec un commit(). Une fois l’erreur identifiée et corrigée, Hubert doit apparaître dans la liste après l’exécution correcte de la requête.

**Question 10**

Après avoir corrigé l'erreur précédente, une nouvelle erreur peut se produire lors de l’ajout de l’agent. Cette situation est courante en développement : la correction d’un problème révèle parfois d'autres bugs sous-jacents. Par exemple, une nouvelle erreur pourrait provenir d’un conflit de clé étrangère ou d’une mauvaise gestion de la connexion à la base de données.

La gestion efficace de ces erreurs permet non seulement de rendre l'application plus robuste, mais aussi de garantir que le système est fiable dans des conditions réelles d'utilisation.

**Question 11**

La fonction de mise à jour du mot de passe est cruciale pour garantir la sécurité du système. Elle permet à un agent de changer son mot de passe régulièrement ou lorsqu’un mot de passe est compromis. Cette mise à jour doit être correctement implémentée pour éviter des erreurs de manipulation des données.

Une fois le mot de passe modifié, il doit être immédiatement enregistré dans la base, et la nouvelle valeur doit remplacer l'ancienne sans laisser de traces. Si la mise à jour échoue (par exemple, en raison d’une connexion perdue avec la base), il est important d’en informer l’utilisateur afin qu’il puisse réessayer.

**Question 12**

Cette étape teste le comportement du système lorsqu’une mise à jour est demandée pour un agent qui n’existe pas dans la base de données. Par défaut, la base devrait soit ignorer la requête, soit renvoyer une erreur contrôlée indiquant que l’agent n’a pas été trouvé. Cela évite que l’application se comporte de manière inattendue et permet aux utilisateurs de comprendre pourquoi la mise à jour n’a pas été effectuée.

Ces tests sont souvent effectués via des scripts de test automatisés, comme ceux fournis dans test\_agents.py. Ils permettent de vérifier que toutes les opérations sont correctement gérées et que le système reste stable en cas d’erreurs.

**Question 13**

Cette étape permet de gérer dynamiquement toutes les entités liées aux incidents. Cela comprend l’ajout, la modification, et la suppression de données sur :

Les attaquants : ajout de nouveaux groupes ou mise à jour de leur sponsor.

Les cibles : ajout ou suppression de secteurs industriels touchés par un incident.

Les incidents eux-mêmes : modification des informations (titre, description, date, etc.) ou mise à jour des groupes suspects impliqués.

Les réponses : ajout d’informations sur la réaction des victimes face aux incidents.

Ces fonctionnalités sont indispensables pour assurer une traçabilité complète de chaque incident. Elles permettent aux agents d’interagir en temps réel avec la base de données, de tenir à jour les informations critiques et de faciliter la planification des réponses aux attaques. Le fait de gérer ces entités de manière centralisée garantit que tous les utilisateurs accèdent à des données précises et cohérentes, ce qui est crucial dans un contexte de sécurité nationale comme celui de la DGSE.

**Question 14**

En examinant le fichier app.py, nous découvrons qu'il sert de point d'entrée pour l'application Flask. Lorsque nous exécutons la commande python app.py, cela initialise le serveur web qui écoute les requêtes entrantes sur le port 5000 de localhost.

Une fois hébergé sur un vrai nom de domaine, les utilisateurs pourront alors effectuer des actions comme récupérer des informations sur les agents, ajouter de nouveaux agents, ou même mettre à jour leurs détails.

**Question 15**

Il s’agit d’une requête HTTP de type GET à l’url suivant : <http://localhost:5000/agents/>. Cela se voit grâce au Blueprint qui permet de rendre un site web facilement modulable, et au décorateur de la fonction qui a comme chemin « / ». Ce même décorateur indique le type de requête.

**Question 16**

Le script exécutant cette requête se trouve à la racine du projet dans le script q16.py. Il renvoie bien le code de réponse 200, signifiant que le site web est bien accessible localement.

**Question 17**

Pour le get\_agent, cela se teste en allant à l’url correspondant. Pour le patch\_password, la commande cURL suivante peut permettre de tester : curl.exe -X PATCH http://localhost:5000/agents/hubert -H "content-type: application/json" -d '{"password": "nouveau\_mot\_de\_passe"}'. Pour la dernière fonction, voici la requête cURL : curl.exe -X POST http://localhost:5000/agents/ -H "content-type: application/json" -d '{"username":"nouvel\_agent","password":"mot\_de\_passe\_secret"}'.

**Question 18**

Pour le get\_agent, cela se teste en allant à l’url correspondant. Pour le patch\_password, la commande cURL suivante peut permettre de tester : curl.exe -X PATCH http://localhost:5000/agents/hubert -H "content-type: application/json" -d '{"password": "nouveau\_mot\_de\_passe"}'. Pour la dernière fonction, voici la requête cURL : curl.exe -X POST http://localhost:5000/agents/ -H "content-type: application/json" -d '{"username":"nouvel\_agent","password":"mot\_de\_passe\_secret"}'.