RAPPORT PROJET PYTHON ELIAS BAROUDI & NASSIM GHLAMI

Table des matières

1.Spécifications

Le programme a pour objectif principal de fournir un moteur de recherche dédié à la cybersécurité, permettant de rechercher des vulnérabilités connues, les CVE (Common Vulnerabilities and Exposures), et d'afficher des solutions théoriques proposées dans des articles scientifiques.

Les fonctionnalités principales, telles que définies par les besoins identifiés, incluent la recherche des CVE à partir de mots-clés. Les CVE retournées doivent contenir des informations détaillées, telles que leur ID, leur description, les notes associées et leur source d'origine. Ensuite, l'application doit afficher les articles scientifiques lorsque ces derniers sont liés aux CVE recherchées. Ces articles doivent être présentés parallèlement aux résultats de recherche, chaque article devant inclure un titre ainsi qu'un lien permettant d'accéder à l'article complet.

De plus, les résultats doivent être triés selon leur pertinence par rapport à la requête de l'utilisateur, en utilisant l'algorithme TF-IDF pour ce faire. L'utilisateur doit également pouvoir filtrer les résultats en fonction de la source des CVE, soit Kevin, soit NST, et choisir s'il souhaite uniquement afficher les CVE associées à des articles scientifiques ou non.

Enfin, une interface utilisateur conviviale est essentielle. L'application propose une interface web simple et claire, permettant de réaliser une recherche, d'afficher les résultats de manière lisible et de sélectionner le nombre de résultats souhaité.

Cette application pourrait s'avérer particulièrement utile pour un professionnel en cybersécurité cherchant des articles scientifiques sur une vulnérabilité spécifique, mais aussi pour un utilisateur moins expérimenté désireux de sécuriser ses applications à l'aide de méthodes fondées sur des travaux scientifiques.

2. L’analyse

2.1 L’environnement de travail et justification

Il a été décidé de travailler avec Python en utilisant des bibliothèques détaillées par la suite, qui permettent de traiter de grandes quantités de données, telles que la fouille de texte ou les calculs statistiques. L’utilisation de Visual Studio Code a été privilégiée afin de faciliter la connexion avec GitHub pour la gestion des commits. Des outils liés à GitHub ont été employés pour garantir un développement fonctionnel et professionnel, en utilisant GitHub Actions pour intégrer Doxygen et pytest.

Pour ce projet, nous avons utilisé plusieurs outils et bibliothèques, tels que :

- Le framework Dash, car il nous fournit une interface web interactive.
- Les bibliothèques NumPy et Pandas, car elles permettent la manipulation des matrices et l’analyse des données.
- Doxygen, pour générer une documentation en ligne sur GitHub Pages.
- Pytest, pour effectuer les tests unitaires.
- L'API Kevin, pour obtenir les CVE les plus récentes.
- L'API NST, pour compléter les informations des CVE.

2.2 les données dans les spécifications

Les principales données identifiées dans les spécifications concernent principalement les objets CVE (provenant de NST et Kevin). Une classe a été créée pour chacune des sources. En ce qui concerne le moteur de recherche et le corpus, ce sont des classes qui ont été conçues tout au long de l’enseignement, durant les TD. Ces classes ont bien sûr été adaptées et perfectionnées pour répondre au contexte spécifique de notre projet.

- Les classes CVE comprennent principalement les éléments suivants :
- Les ID des CVE
- Les descriptions des CVE
- Les dates d’ajout des CVE
- Les notes associées aux CVE

Ces objets sont instanciés dans la partie principale du script lorsque nous envoyons une requête aux API pour récupérer les objets.

Le **corpus** est la structure qui organise toutes les CVE afin de créer un support sur lequel le moteur de recherche pourra travailler. Il joue un rôle clé dans les calculs de TF-IDF. Il est composé de :

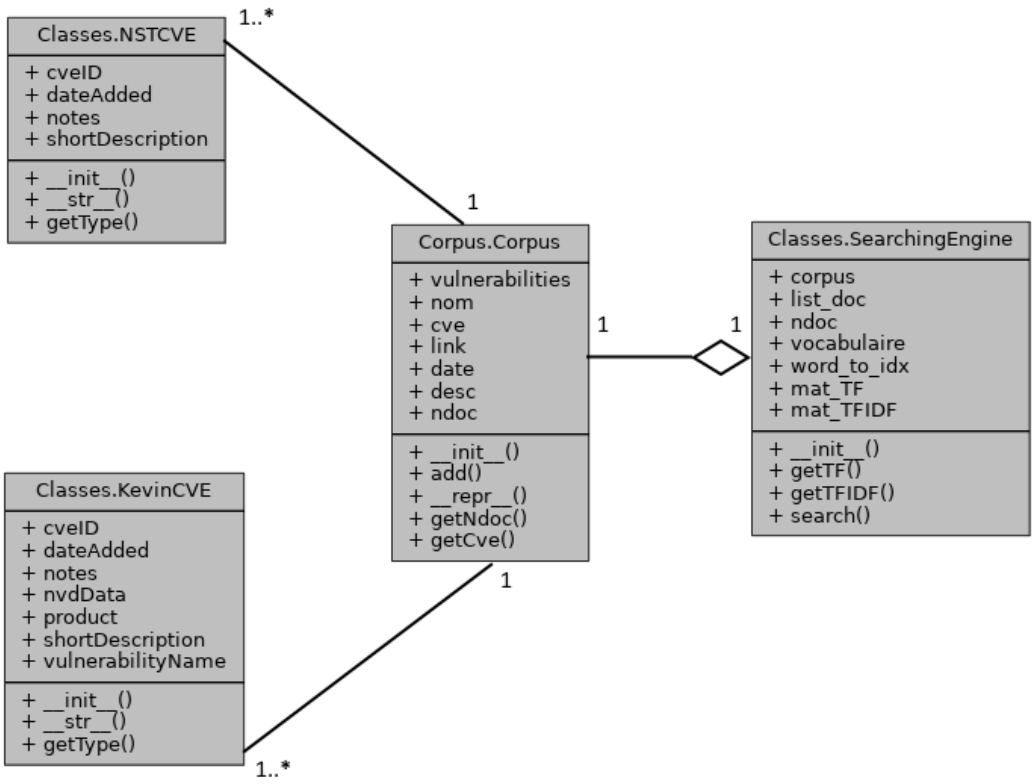
- La liste des CVE
- Des mots-clés
- Le nombre de documents

En entrée, nous avons les **requêtes utilisateur**, qui correspondent aux données saisies ou sélectionnées par l'utilisateur pour faciliter sa recherche :

- Les mots-clés
- Les filtres (source, nombre de résultats, articles associés)

En sortie, nous avons la classe du **moteur de recherche**, qui renvoie un DataFrame à l'aide de la fonction MotRecherch.search() avec certains paramètres qui seront détaillés plus tard dans le rapport. Ce DataFrame comprend la liste des documents jugés pertinents selon la requête, calculée à partir du TF-IDF et de la similarité cosinus.

2.3 Diagramme des classes



Voici le diagramme des classes de l'application :

- **NSTCVE** et **KevinCVE** sont en **association** avec le **Corpus**. Plusieurs objets **CVE** peuvent être associés à un seul **Corpus**.
- Le **Corpus** est en **agrégation** avec la classe **SearchEngine** : le moteur de recherche est créé à partir du **Corpus**. Un **Corpus** est associé à un seul **SearchEngine**.

2.3.2 Classe KevinCVE

C'est une sous-classe de **CVE** qui représente les CVE provenant de l'API Kevin.

Attributs :

- **cveID**
- **dateAdded**
- **notes**
- **nvdData**
- **product**
- **shortDescription**
- **vulnerabilityName**

Aucune méthode pertinente, à l'exception de la méthode __str__ pour la représentation sous forme de chaîne, ainsi que getType(), qui retourne le type de l'objet nécessaire à l'affichage dans l'interface.

2.3.3 Classe NSTCVE

C'est également une sous-classe de **CVE**, représentant les CVE issues de l'API **NST**.

Attributs :

- **cveID**
- **dateAdded**
- **notes**
- **shortDescription**

Tout comme **NSTCVE**, il n'y a aucune méthode pertinente à l'exception de la méthode __str__ pour la représentation sous forme de chaîne, ainsi que getType(), qui retourne le type de l'objet nécessaire à l'affichage dans l'interface.

2.3.4 La classe Corpus

Cette classe gère un ensemble de CVE et les organise de manière à permettre des recherches efficaces. Elle dispose de plusieurs attributs principaux :

Attributs :

- **nom**
- **cve** : comprend la liste des objets **CVENST** et **CVEKevin**
- **link** : contient la liste des liens vers les articles Arxiv, associés à l'ID des CVE
- **ndoc** : nombre de documents
- **date**
- **desc**
- **vulnerabilities** : liste de mots-clés représentant les vulnérabilités

Méthodes :

- **Add(self, doc)** : Cette méthode permet d'ajouter un objet au corpus. Elle effectue également une recherche sur Arxiv pour fournir des articles scientifiques relatifs à la vulnérabilité. Un algorithme de sélection de mots-clés dans la description est utilisé, en se basant sur l'attribut **vulnerabilities** de la classe **Corpus**. Lorsqu'un mot de la description de la vulnérabilité ajoutée est trouvé dans la liste des mots-clés, il est ajouté à la requête formulée pour Arxiv. Si cette méthode échoue, notamment pour un objet de type **CVE NST**, on récupère les trois derniers mots du nom de la vulnérabilité (généralement très caractéristiques de la vulnérabilité). La requête est alors envoyée à Arxiv via l'API, et les informations récupérées sont stockées dans le dictionnaire **link**, avec l'URL de l'article comme valeur et l'ID de la CVE comme clé. Le paramètre **doc** correspond à la CVE que l'on souhaite ajouter.
- **getNdoc()** : Cette méthode permet d'obtenir le nombre de documents dans le corpus, nécessaire pour l'affichage et pour les opérations sur le corpus.
- **getCVE()** : Cette méthode retourne la liste des CVE, nécessaire pour l'affichage et les opérations sur le corpus.

2.3.5 La classe SearchingEngine

Cette classe implémente le moteur de recherche.

Elle dispose de plusieurs attributs principaux :

Attributs :

- **corpus** : Le corpus contenant les CVE.
- **word_to_idx** : Un dictionnaire qui associe les mots à leurs indices.
- **vocabulaire** : Liste des mots apparaissant dans toutes les descriptions.
- **list_doc** : Liste des CVE présentes dans le corpus.
- **mat_TF** : Matrice TF (Term Frequency).
- **mat_TFIDF** : Matrice TF-IDF.

La classe comprend également plusieurs variables intermédiaires permettant de construire les matrices **TF** et **TF-IDF**.

Méthodes:

- **getTf()** : Cette méthode calcule la matrice **TF** pour le corpus.
- **getTFIDF()** : Cette méthode calcule la matrice **TF-IDF** pour le corpus.
- **search(query, num_results, filters, has_articles)** : Cette méthode effectue une recherche basée sur une requête et retourne les résultats les plus pertinents. Pour ce faire, les matrices **TF-IDF** sont recalculées sur le vecteur de la requête. Ensuite, un calcul de similarité cosinus est effectué entre le vecteur de la requête et la matrice **TF-IDF** de chaque document. La méthode retourne une liste triée des documents les plus pertinents en fonction de cette mesure de similarité.

2.3.6 Relations d'héritage

Dans une première version du projet, il avait été envisagé d'utiliser une classe **CVE** pour regrouper les classes **CVENST** et **CVEKevin**. Cependant, aucune utilisation pertinente de la classe **CVE** n'a été identifiée au cours du développement de l'application. Il a donc été décidé de mettre cette idée de côté et de se concentrer sur le développement de deux classes distinctes, chacune regroupant les informations les plus importantes en fonction de la source.

2.3.7 Relations d’utilisations

La classe **Corpus** gère les collections d'objets **CVENST** et **CVEKevin**. Elle sert de base structurée pour stocker et organiser les données.

La classe **SearchEngine** utilise les données du **Corpus** pour créer les matrices **TF** et **TF-IDF**, qui permettent d'effectuer des recherches pertinentes. Elle est également chargée de réaliser ces recherches en fonction des entrées de l’utilisateur.

Enfin, l'interface utilisateur, définie dans le fichier **interface.py**, interagit avec **SearchEngine**. Elle envoie les requêtes de recherche, récupère les résultats et les affiche. Il est important de noter que l'instanciation des objets à partir de ces classes se fait via le fichier **main.py**, dans lequel on retrouve l'appel aux API, la possibilité de charger des variables sauvegardées et la création des objets.

3. La conception

3.1 partages des tâches

Tout d'abord, avant de soumettre la version 1 de l'application, un travail collectif a été réalisé lors des deux dernières séances de TD. Ce travail a permis de définir les spécifications, d'imaginer l'interface et le fonctionnement de l'application, ainsi que de mettre en place le plan de développement.

Ensuite, Elias a repris les bases établies durant les deux derniers TD pour développer les versions 1 et 2, correspondant aux TDs 3 à 7. Il a ainsi posé les fondations de l'application ainsi que le moteur de recherche.

De son côté, Nassim a travaillé sur la version 3, qui inclut l'interface et les extensions, et a rédigé le rapport.

Les échanges au sein de l’équipe se faisaient principalement via WhatsApp à chaque avancement du développement. Un commit sur GitHub était également effectué afin que chaque membre du groupe puisse consulter les progrès réalisés et apporter des retours si nécessaire.

3.2 Détail Algorithme

Algorithme de récupération des CVE :

L'algorithme de récupération des CVE est lancé dès le démarrage de l'application. Il est représenté par les fonctions init() et getCorpus() dans le fichier **main.py**. Ce processus comprend plusieurs étapes :

1. **Requêtes aux API (NST et Kevin)** : Les requêtes sont effectuées séparément auprès des APIs NST et Kevin.
2. **Vérification de la présence de fichiers sauvegardés** : Le module **os** est utilisé pour vérifier l'existence des fichiers data.pkl et corpus.pkl. Si ces fichiers ne sont pas trouvés, le script suit un flux alternatif pour télécharger les données depuis les API Kevin et NST et créer le corpus. En parallèle, des articles scientifiques sont récupérés sur Arxiv.
3. **Sauvegarde des données** : Étant donné que ces traitements peuvent être longs, il a été décidé de sauvegarder à l'avance ces données dans des fichiers .pkl pour accélérer les exécutions futures. Si les fichiers existent déjà, le script charge simplement les données à l'aide du module **pickle**.
4. **Retour des données** : Enfin, un objet collection contenant la liste des CVE est envoyé à l'utilisateur, ou, dans le cas de la méthode getCorpus(), le **corpus** est renvoyé.

Algorithme de sélection des articles scientifiques :

Cet algorithme est sollicité lors de l'ajout d'un document au **corpus**, via la méthode Corpus.add(doc). Il permet d'extraire des mots-clés pertinents à partir de la description d'une CVE pour sélectionner des articles scientifiques en lien avec cette vulnérabilité.

1. **Extraction de mots-clés** : L'algorithme parcourt l'attribut shortDescription de l'objet doc (qui peut être du type **NSTCVE** ou **KevinCVE**) et cherche des correspondances avec une liste de mots-clés définie dans la classe **Corpus**. Ces mots-clés concernent les vulnérabilités connues en cybersécurité.
2. **Requête à Arxiv** : Si un mot-clé est trouvé dans la description de la CVE, il est ajouté à la requête à l'API d'Arxiv. Si l'algorithme basé sur les mots-clés n'est pas suffisant et que l'objet est de type **KevinCVE**, les trois derniers mots du nom de la vulnérabilité (vulnerabilityName) sont extraits et ajoutés à la requête.
3. **Traitement de la requête** : La requête est transformée en minuscules et les mots sont séparés par des signes +, ce qui est nécessaire pour interroger l'API d'Arxiv.
4. **Récupération des résultats** : Une fois la requête envoyée, l'API d'Arxiv retourne un ensemble d'articles scientifiques. Seules les URLs des articles au format PDF sont

récupérées et stockées dans le dictionnaire **link** de la classe **Corpus**, avec l'ID de la CVE correspondant à la clé.

Algorithme de calcul des matrices TF-IDF :

Le calcul des matrices **TF-IDF** a été développé principalement durant les séances de TD, en dehors du développement de l'application elle-même. Cet algorithme repose sur les questions fournies lors des TD.

1. **Calcul de la matrice TF** : La matrice **TF** est construite en parcourant toutes les descriptions des CVE dans le corpus et en répertoriant les mots dans un set (pour éviter les doublons). Pour chaque document, la fréquence d'apparition de chaque mot est calculée. Cela donne une matrice **TF**, où chaque colonne correspond à un mot du vocabulaire et chaque ligne à un document, avec les fréquences des mots dans les documents.
2. **Calcul de la matrice TF-IDF** : La matrice **TF-IDF** est ensuite calculée en combinant la matrice **TF** avec la formule IDF :

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

Avec :

- **tf** : le calcul TF pour chaque document
- **N** : pour ne nombre de doc dans le coprus
- **Df** : pour le nombre de document ou le terme apparait

Cela donne une matrice **TF-IDF** où chaque document est représenté par une liste de mots avec leurs valeurs **TF-IDF**.

Algorithme de recherche :

L'algorithme de recherche est activé lors de l'appel à la méthode MotRecherch.search(). Cette méthode prend en paramètres la requête de l'utilisateur, le nombre de résultats à retourner, ainsi que des filtres (par exemple, pour limiter les résultats à ceux contenant des articles scientifiques).

1. **Transformation de la requête** : La requête de l'utilisateur est récupérée sous forme de chaîne de caractères (par exemple, "sql injection"). Cette chaîne est ensuite transformée en un vecteur de taille égale au vocabulaire du corpus, où chaque élément correspond à un mot de la requête. Les éléments du vecteur qui ne sont pas des mots de la requête sont initialisés à zéro.

2. **Calcul de la similarité** : Un calcul **TF** est effectué sur ce vecteur de requête, suivi d'un calcul **TF-IDF**. Ensuite, la similarité cosinus entre le vecteur de la requête et les matrices **TF-IDF** de chaque document est calculée. La similarité est calculée à l'aide de la formule suivante :

$$\cos\theta = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}.$$

Cette opération donne une mesure de similarité entre chaque document et la requête de l'utilisateur.

3. **Tri des résultats** : La similarité cosinus est calculée pour chaque document du corpus. Les résultats sont ensuite triés de manière décroissante en fonction de leur similarité.
4. **Retour des résultats** : Enfin, les documents les plus pertinents, triés selon leur similarité avec la requête, sont récupérés par leur ID et renvoyés sous la forme d'un **DataFrame**.

3.3 Problèmes rencontrés

Le premier défi auquel nous avons fait face a été le choix de la direction du projet. Bien que nous ayons travaillé sur des TD en cours, cela ne suffisait pas. Nous voulions éviter de simplement reproduire ce qui avait déjà été fait. Nous avons donc opté pour un projet qui nous serait utile à long terme. Comme nous nous orientons tous les deux vers la cybersécurité, il nous a semblé logique de choisir un projet lié à ce domaine.

Le deuxième défi a concerné la recherche des API adéquates. Nous avons exploré plusieurs options, mais certaines étaient payantes ou trop complexes à utiliser. Après plusieurs tentatives infructueuses, nous avons finalement choisi les API Kevin ainsi que NST qui se sont avérées être les plus adaptées à nos besoins.

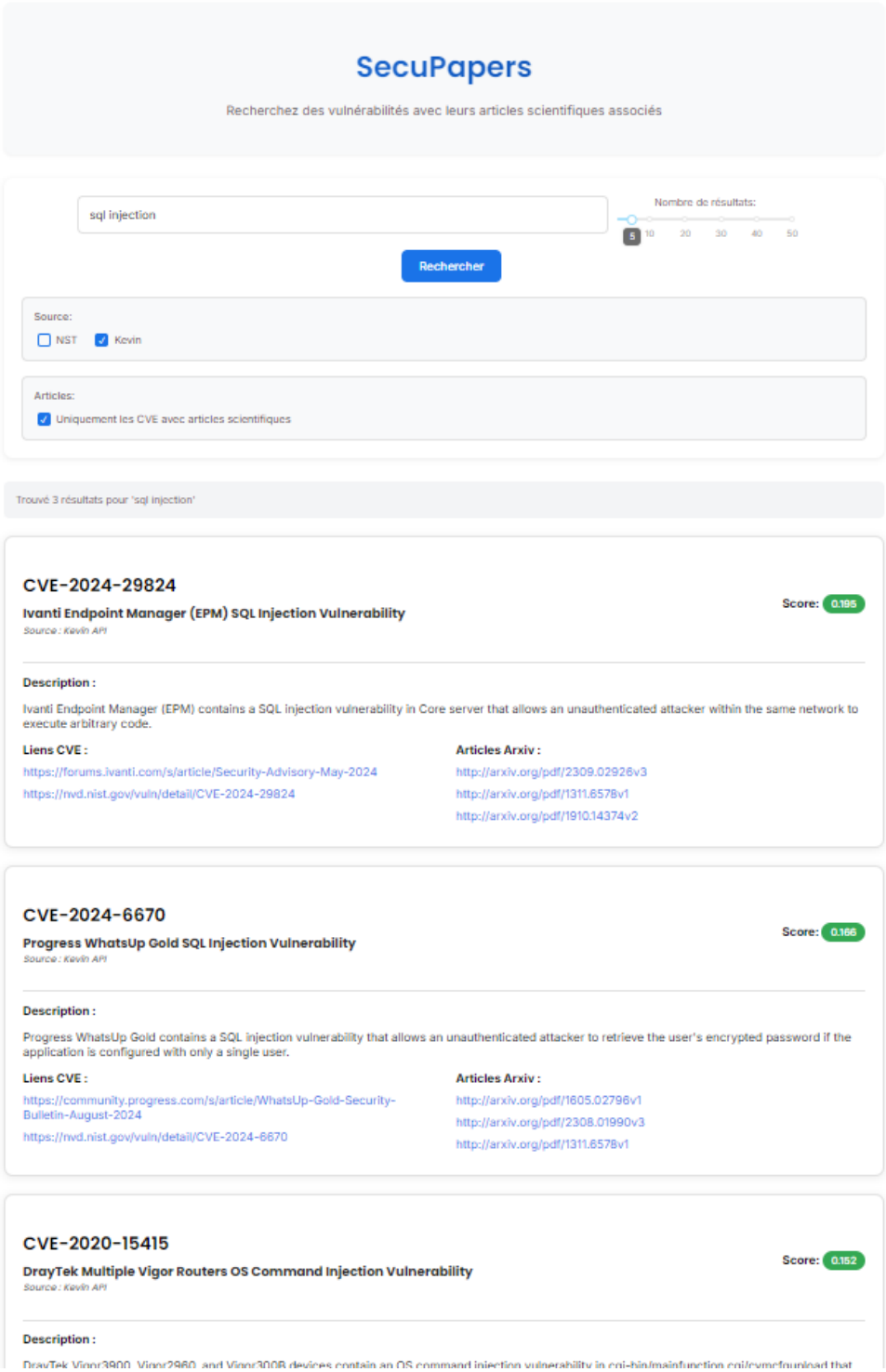
Enfin, un autre problème est survenu lors du développement de l'interface utilisateur. Initialement, il était prévu d'utiliser Jupyter Notebook pour concevoir l'interface. Cependant, nous avons rapidement constaté que ce support ne permettait pas d'exploiter pleinement les capacités de l'application. Nous avons donc pris la décision de passer à **Dash**, ce qui nous a permis de créer une interface web interactive et plus fonctionnelle, adaptée aux exigences du projet.

3.4 Exemple commenté d'utilisation du programme

Voici un aperçu de la zone de recherche. L'utilisateur peut spécifier le nombre de documents à retourner, saisir des mots-clés, puis lancer la recherche en cliquant sur le bouton "**Rechercher**". L'utilisateur peut également filtrer les résultats :

- Choisir les sources qui l'intéressent (NST, Kevin)
- Choisir uniquement les documents pour lesquels un article scientifique a été trouvé

Voici un aperçu de l'interface après l'exécution d'une recherche. Elle affiche les différentes CVE jugées pertinentes en fonction des critères de recherche, avec les informations suivantes : le score de pertinence, le nom de la CVE, son ID, des liens vers des informations détaillées sur la CVE, ainsi que les articles scientifiques associés à la vulnérabilité.



L'utilisateur peut désormais cliquer sur les liens CVE, ou les articles Arxiv pour avoir les pdf des articles scientifiques associées aux vulnérabilités.

4. La validation

4.1 Test unitaires

Les tests unitaires ont été réalisés pour vérifier les méthodes principales impliquées dans le fonctionnement de l'application. Ils couvrent des opérations élémentaires telles que l'instanciation des classes, l'instanciation du moteur de recherche, l'ajout des objets CVE au corpus et les tests de la méthode de recherche du moteur de recherche. Ces tests ont été effectués à l'aide de **pytest**, en exécutant le fichier **test.py**. Ce fichier est disponible dans le repository et les résultats des tests peuvent être consultés dans les GitHub Actions du repository.

Voici un exemple de fonction de test unitaire (le reste des fonctions est consultable dans le fichier **test.py** du repository de l'application).

Cette fonction teste le bon fonctionnement du moteur de recherche :

1. Génère une requête avec le filtre de source
2. S'assure que le résultat de la fonction est bien un DataFrame contenant des résultats pour kevin
3. Verifie les résultats contenus dans le DataFrame
4. Même processus pour la source NST

```
def test_search_engine_search(sample_search_engine):
    # Test de recherche avec seulement la source Kevin
    results_kevin = sample_search_engine.search("security vulnerability", 2, ["Kevin"], False)
    assert isinstance(results_kevin, pd.DataFrame)
    if not results_kevin.empty:  # Vérifie si des résultats ont été trouvés
        assert all(col in results_kevin.columns for col in ['Source', 'CVE ID', 'Description', 'Arxiv related'])
        assert all(row['Source'] == 'Kevin API' for _, row in results_kevin.iterrows())

    # Test de recherche avec seulement la source NST
    results_nst = sample_search_engine.search("security vulnerability", 2, ["NST"], False)
    assert isinstance(results_nst, pd.DataFrame)
    if not results_nst.empty:  # Vérifie si des résultats ont été trouvés
        assert all(col in results_nst.columns for col in ['Source', 'CVE ID', 'Description', 'Arxiv related'])
```

```
assert all(row['Source'] == 'NST API' for _, row in results_nst.iterrows())
```

Il est important d'informer les correcteurs que ces fonctions de tests ont été perfectionnées avec l'aide de ChatGPT. Les fonctions à tester ont été définies à l'avance, et un prototype des tests avait déjà été créé avant l'utilisation de l'intelligence artificielle. ChatGPT a simplement permis de perfectionner ces fonctions de tests unitaires, les rendant ainsi plus concises et mieux structurées.

4.2 Test Globaux

Les tests globaux ont été réalisés à la fin du projet, une fois que toutes les fonctionnalités étaient en place. L'aspect principal testé était la zone de recherche dans l'interface Dash. Nos tests consistaient principalement à ajouter ou retirer les filtres proposés, ajuster le slider pour le nombre de résultats à afficher, tester les URL fournies par Arxiv et vérifier leur correspondance avec la vulnérabilité, ainsi qu'à cliquer sur les notes des CVE. Nous avons aussi jugé de la pertinence des documents retournés en fonction des requêtes effectuées. Voici quelques exemples afin de tester les requêtes : ('sql injection', 'cross-site scripting', 'buffer overflow', 'Windows', 'Linux')

L'application fonctionne correctement et réagit comme prévu en fonction des filtres et des requêtes de l'utilisateur. Le nombre de résultats est bien ajusté par le slider. Les résultats concernant les vulnérabilités et les articles scientifiques associés sont également très cohérents. En somme, l'application permet à n'importe quel utilisateur de trouver des CVE et des articles scientifiques correspondants à ses mots-clés.

Il convient de noter qu'un manque de données sur les articles scientifiques fournis peut être observé, bien qu'il ne s'agisse pas d'un problème en soi. En effet, dans environ 30 % des cas, aucune référence à un article scientifique Arxiv n'est disponible pour une CVE donnée. Cela pourrait être dû à une limitation dans la disponibilité des articles scientifiques. À l'avenir, il serait intéressant d'envisager l'implémentation d'une autre source d'articles scientifiques, comme Google Scholar, pour compléter les résultats.

5. La maintenance

Il serait intéressant d'envisager une version multilingue de l'application permettant aux utilisateurs de rechercher des CVE ou des articles scientifiques dans différentes langues. Pour cela, nous pourrions utiliser des bibliothèques telles que l'API Google Translate pour traduire les requêtes des utilisateurs. En parallèle, il serait nécessaire d'élargir le corpus en incluant des descriptions de CVE traduites. Bien que cette approche soit de difficulté moyenne, car les bibliothèques de traduction sont bien intégrées à Python, il serait toutefois nécessaire de gérer les erreurs de traduction, surtout pour des termes techniques qui peuvent être plus complexes à traiter.

Une autre fonctionnalité intéressante serait de permettre aux utilisateurs d'exporter les résultats de recherche sous forme de fichiers CSV ou PDF. Pour cela, nous pourrions utiliser la bibliothèque Pandas pour l'exportation en CSV et FPDF pour la création de fichiers PDF. Cette fonctionnalité serait relativement facile à implémenter car ces bibliothèques sont largement utilisées et bien documentées.

Il serait également pertinent d'enrichir l'affichage des informations concernant les CVE, en offrant la possibilité de trier les résultats par critères tels que la criticité, le vecteur d'attaque ou même le statut de la vulnérabilité. Cette amélioration consisterait essentiellement à étendre les capacités de l'interface utilisateur, car les CVE récupérées par les API contiennent déjà ces informations.

Enfin, une amélioration notable pourrait concerner les API utilisées pour la récupération des articles scientifiques. Bien que l'API Arxiv fonctionne de manière satisfaisante, il serait intéressant d'introduire des sources supplémentaires, telles que Google Scholar, afin de diversifier les résultats. Cette intégration serait de difficulté moyenne : il faudrait d'abord comprendre le fonctionnement de l'API de Google Scholar ou d'une autre source similaire, puis implémenter la récupération des articles et intégrer ces nouvelles données dans le contexte de notre application.

6. Liens

Comme mentionné dans le rapport, l'application bénéficie d'une intégration continue et suit des pratiques modernes de développement d'applications via GitHub. Une documentation a été générée à l'aide de Doxygen, et un repository GitHub est disponible, comprenant les trois versions de l'application (v1, v2, v3) sous forme de *releases*. Ces versions permettent au correcteur de télécharger et consulter l'application dans ses différentes étapes de développement, de la v1 à la v3.

Ce repository et la documentation générée via Doxygen fournissent des informations supplémentaires pertinentes sur l'application, telles que des détails concernant les classes, leurs attributs et leurs méthodes, ce qui peut compléter les informations mentionnées dans ce rapport.

Repository GitHub : https://github.com/EliasBaroudi/search-engine-python/tree/main

Gh-Pages (Doxygen) : https://eliasbaroudi.github.io/search-engine-python/html/index.html