IPAnalyzer.py

```python
#!/usr/bin/env python3
# encoding: utf-8

# Date : novembre 2024
# Version : 1.0
# Auteur : Nicolas Clerbout

# Description : Analyzer Cortex pour enrichir des observables IP sur TheHive à partir de sources externes
# Requiert des clés API - les comptes non payants ont des limitations sur le nombre d'utilisations
# Autres informations (licence, etc) et configuration dans le fichier IPAnalyzer.json

import requests
from cortexutils.analyzer import Analyzer


class IPAnalyzer(Analyzer):

    def __init__(self):
        '''
        Récupère des données de configuration pour spécifier les sources consultées

        Les sources sont les entrées d'un dict self.sources et sont caractérisées
        par une clé API, une url et le header d'identification par clé API nécessaire
        aux requêtes HTTP

        url et header sont définis au moyen de fonctions lambda pour s'appliquer à différentes IP
        et clés API

        En cas de changements (ajouts/suppressions) de sources, respecter la convention
        self.sources = {
            "<Source>": {
                "api_key": <value>,
                "url": <value>,
                "headers": <value>
            }
        }
        car les fonctions fetch_data_from_source() et run() sont définie en supposant cette
        construction
        '''

        super().__init__()

        self.sources = {
            "VirusTotal": {
                "api_key": self.get_param("config.virustotal_api_key", None, "Clé API manquante (VirusTotal)"),
                "url": lambda ip: f"{self.get_param('config.virustotal_api_url',
                                                    'https://www.virustotal.com/api/v3/ip_addresses/')}{ip}",
                "headers": lambda key: {"x-apikey": key}
            },
            "OTX_Alienvault": {
                "api_key": self.get_param("config.otx_api_key", None, "Clé API manquante (OTX-Alienvault)"),
                "url": lambda ip: f"{self.get_param('config.otx_api_url',
                                                    'https://otx.alienvault.com/api/v1/indicators/IPv4/')}{ip}/general",
                "headers": lambda key: {"X-OTX-API-KEY": key}
            }
        }

    def fetch_data_from_source(self, source, ip_address):
        '''
        Utilise self.sources et requests pour interroger les sources via leur API

        :param source: Clé dans le dict self.sources
        :param ip_address: IP analysée (string) - sera récupérée dans run()
        :return: objet JSON avec les réponses des sources
        '''

        source_params = self.sources.get(source)
        if not source_params:
            self.error(f"Source inconnue : {source}")

        api_key = source_params["api_key"]
        url = source_params["url"](ip_address)
        headers = source_params["headers"](api_key)

        try:
            response = requests.get(url, headers=headers)
            response.raise_for_status()
            return response.json()

        except requests.RequestException as e:
            self.error(f"Erreur récupération de données depuis {source}: {e}")

    def verdict_VirusTotal(self, details):
        '''
        Analyse les résultats de la requête à l'API de VirusTotal

        Vérifie la présence des champs pertinents dans la réponse de VirusTotal et les utilise pour
        enrichir l'IP analysée

        Adaptable pour d'autres sources si self.sources est modifié (ajout ou suppression de sources)
        Attention : bien utiliser le même nom que la clé correspondante dans self.sources dans le nom
        de la fonction pour que le nom verdict_method dans run() soit correctement défini

        :param details: dict - correspond à la réponse de fetch_data_from_source() pour VirusTotal
        :return: tuple (verdict, report_url, country) à utiliser dans les actions déclenchées sur
        TheHive par l'analyzer
        '''

        data = details.get("data")
        attributes = data.get("attributes")
        stats = attributes.get("last_analysis_stats")
        country = attributes.get("country")

        if not data:
            self.error("Pas de champ 'data' dans la réponse de VirusTotal")
        elif not attributes:
            self.error("Le champ 'data' n'a pas de champ 'attributes' dans la réponse de VirusTotal")
        elif not stats:
            self.error("Le champ 'attributes' n'a pas de champ 'last_analysis_stats' dans la réponse de VirusTotal")

        else:
            if stats.get("malicious", 0) > 0:
                verdict = "Malicious"

            elif stats.get("suspicious", 0) > 0:
                verdict = "Suspicious"

            else:
                verdict = "Non-Malicious"

            report_id = details["data"]["id"]
            report_url = f"https://www.virustotal.com/gui/ip-address/{report_id}"
            country = country

            return verdict, report_url, country

    def verdict_OTX_Alienvault(self, details):
        '''
        Analyse les résultats de la requête à l'API de OTX_Alienvault

        Vérifie la présence des champs pertinents dans la réponse d'OTX et les utilise pour enrichir
        l'IP analysée

        Adaptable pour d'autres sources si self.sources est modifié (ajout ou suppression de sources)
        Attention : bien utiliser le même nom que la clé correspondante dans self.sources dans le nom
        de la fonction pour que le nom verdict_method dans run() soit correctement défini

        :param details: dict - correspond au résultat de fetch_data_from_source() pour OTX
        :return: tuple (verdict, report_url, country) à utiliser dans les actions déclenchées sur
        TheHive par l'analyzer
        '''

        pulse_info = details.get("pulse_info")
        if not pulse_info:
            self.error("Pas de champ 'pulse_info' dans la réponse d'OTX")

        else:
            if pulse_info.get("count", 0) > 1:
                verdict = "Malicious"

            elif pulse_info.get("count", 0) == 1:
                verdict = "Suspicious"

            else:
                verdict = "Non-Malicious"

            report_id = details.get("indicator", 0)
            report_url = f"https://otx.alienvault.com/indicator/ip/{report_id}"
            country = details.get("country_code")

            return verdict, report_url, country

    def summary(self, raw):
        '''
        Présente les verdicts des différentes sources de manière visuelle

        Construit une taxonomie par source avec un code couleur ('level') correspondant au verdict,
        le nom de l'analyzer ('namespace'), la source ('source') et le verdict ('verdict')
        Tient compte de la structure du dict results construit dans la fonction run() : ignore la Clé
        "Actions" qui concerne la fonction operations()

        :param raw: dict construit par run()
        :return: dict où la valeur est la liste des taxonomies construites
        '''

        taxonomies = []
        namespace = "IPAnalyzer"

        for source, result in raw.items():
            if source == "Actions":
                continue

            verdict_key = f"Verdict {source}"
            if verdict_key in result:
                verdict = result[verdict_key]
            else:
                self.error(f"Pas de verdict pour la source '{source}'")

            if verdict == "Malicious":
                level = "malicious"
            elif verdict == "Suspicious":
                level = "suspicious"
            else:
                level = "safe"

            taxonomies.append(self.build_taxonomy(level, namespace, source, verdict))

        return {"taxonomies": taxonomies}

    def artifacts(self, raw):
        '''
        Réécrit la méthode artifacts() de la classe Analyzer car cet analyzer ne génère pas de nouvel
        observable
        '''

        return []

    # Les 3 fonctions suivantes gèrent les actions que l'analyzer peut être amené à réaliser sur
    # TheHive en fonction des résultats de l'analyse. Elles permettent d'éviter les erreurs dans
    # l'utilisation de self.build_operation() selon les cas tout en étant explicite sur les actions
    # réalisées

    def create_task(self, task):
        return self.build_operation("CreateTask", **task)

    def add_case_tag(self, label):
        return self.build_operation("AddTagToCase", tag=label)

    def add_artifact_tag(self, label):
        return self.build_operation("AddTagToArtifact", tag=label)

    def operations(self, raw):
        '''
        Construit la liste des opérations (actions) réalisées sur TheHive par l'analyzer

        Les actions sont effectivement réalisées dans run(), regroupées en une liste elle-même
        incluse dans les resultats sous la clé "Actions"

        :param raw: dict - en pratique sera le dictionnaire results défini dans run()
        :return: list operations
        '''

        operations = []

        if raw.get("Actions"):
            operations = operations + raw["Actions"]

        return operations

    def run(self):
        '''
        Utilise les fonctions antérieures pour enrichir l'observable et le case si pertinent

        Récupère l'IP à analyser ;
        Regroupe les données fournies par les sources de self.sources sur l'IP dans le dict results ;
        Utilise les fonctions verdict_{source} pour qualifier l'IP ;
        Crée les tâches et tags (pour l'observable et le case) pertinents si l'IP est malicieuse ou
        suspecte ;
        Ajoute la liste des actions réalisées dans TheHive au dict results

        "results" aura la forme générale :
        {
            "Source": {
                fetch_data_from_source(Source, ip),                     (--> est un dict)
                "Verdict Source": "string"
                "Report Source": "string"
                "Country Source": "string"
            },
            "Actions": list [action 1, action 2...]
        }

        :return: application de la méthode report() de la classe Analyzer au dict results
        '''

        ip_address = self.get_data()
        results = {}
        malicious_sources = []
        suspicious_sources = []
        ops_done = []

        for source in self.sources.keys():
            data = self.fetch_data_from_source(source, ip_address)
            results[source] = {"data": data}

            try:
                verdict_method = getattr(self, f"verdict_{source}")
                verdict, report_url, country = verdict_method(data)
                results[source][f"Verdict {source}"] = verdict
                results[source][f"Report {source}"] = report_url
                results[source][f"Country {source}"] = country

                if verdict == "Malicious":
                    malicious_sources.append(source)

                if verdict == "Suspicious":
                    suspicious_sources.append(source)

            except AttributeError:
                self.error(f"Pas de méthode de verdict trouvée pour '{source}'")

        if malicious_sources or suspicious_sources:

            vt_report = results["VirusTotal"].get("Report VirusTotal")
            otx_report = results["OTX_Alienvault"].get("Report OTX_Alienvault")

            new_task = {
                "title": f"Consulter les détails concernant l'ip {ip_address}",
                "description": "Voir les rapports :\n\n"
                f"VirusTotal : {vt_report}\n\n"
                f"OTX-Alienvault : {otx_report}",
                "status": "Waiting"
            }
            task_op = self.create_task(new_task)
            ops_done.append(task_op)

            new_task_1 = {
                "title": f"Bloquer le trafic réseau vers et depuis {ip_address}",
                "description": "Configurer l'IDS/IPS et le firewall pour bloquer le trafic dangereux",
                "status": "Waiting"
            }
            task_op_1 = self.create_task(new_task_1)
            ops_done.append(task_op_1)

        artifact_tag = results["VirusTotal"]["Country VirusTotal"]
        art_tag_op = self.add_artifact_tag(artifact_tag)
        ops_done.append(art_tag_op)

        artifact_tag_2 = "cortex analyzer"
        art_tag_op_2 = self.add_artifact_tag(artifact_tag_2)
        ops_done.append(art_tag_op_2)

        case_tag = "IP analyzer"
        case_tag_op = self.add_case_tag(case_tag)
        ops_done.append(case_tag_op)

        results["Actions"] = ops_done
        self.report(results)
```

```
if __name__ == "__main__":
    IPAnalyzer().run()
```

HashAnalyzer.py

```python
#!/usr/bin/env python3
# encoding: utf-8

# Date : novembre 2024
# Version : 1.0
# Auteur : Nicolas Clerbout

# Description : Analyzer Cortex pour enrichir des observables hash sur TheHive à partir de sources externes
# Requiert des clés API - les comptes non payants ont des limitations sur le nombre d'utilisations
# Autres informations (licence, etc) et configuration dans le fichier HashAnalyzer.json

import requests
from cortexutils.analyzer import Analyzer


class HashAnalyzer(Analyzer):

    def __init__(self):
        '''
        Récupère des données de configuration pour spécifier les sources consultées

        Les sources sont les entrées d'un dict self.sources et sont caractérisées
        par une clé API, une url et le header d'identification par clé API nécessaire
        aux requêtes HTTP

        url et header sont définis au moyen de fonctions lambda pour s'appliquer à différents hashes
        et clés API

        En cas de changements (ajouts/suppressions) de sources, respecter la convention
        self.sources = {
            "<Source>": {
                "api_key": <value>,
                "url": <value>,
                "headers": <value>
            }
        }
        car les fonctions fetch_data_from_source() et run() sont définie en supposant cette
        construction
        '''

        super().__init__()

        self.sources = {
            "VirusTotal": {
                "api_key": self.get_param("config.virustotal_api_key", None, "Missing VirusTotal API Key."),
                "url": lambda hash: f"{self.get_param('config.virustotal_api_url',
                                                     'https://www.virustotal.com/api/v3/files/')}{hash}",
                "headers": lambda key: {"x-apikey": key}
            },
            "OTX_Alienvault": {
                "api_key": self.get_param("config.otx_api_key", None, "Missing OTX-Alienvault API Key."),
                "url": lambda hash: f"{self.get_param('config.otx_api_url',
                                               'https://otx.alienvault.com/api/v1/indicators/file/')}{hash}/general",
                "headers": lambda key: {"X-OTX-API-KEY": key}
            }
        }

    def fetch_data_from_source(self, source, hash_value):
        '''
        Utilise self.sources et requests pour interroger les sources via leur API

        :param source: Clé dans le dict self.sources
        :param hash_value: hash analysé (string) - sera récupéré dans run()
        :return: objet JSON avec les réponses des sources
        '''

        source_params = self.sources.get(source)
        if not source_params:
            self.error(f"Source inconnue : {source}")

        api_key = source_params["api_key"]
        url = source_params["url"](hash_value)
        headers = source_params["headers"](api_key)

        try:
            response = requests.get(url, headers=headers)
            response.raise_for_status()
            return response.json()
        except requests.RequestException as e:
            self.error(f"Erreur de récupération données depuis {source}: {e}")

    def verdict_VirusTotal(self, details):
        '''
        Analyse les résultats de la requête à l'API de VirusTotal

        Vérifie la présence des champs pertinents dans la réponse de VirusTotal et les utilise pour
        enrichir le hash analysé

        Adaptable pour d'autres sources si self.sources est modifié (ajout ou suppression de sources)
        Attention : bien utiliser le même nom que la clé correspondante dans self.sources dans le nom
        de la fonction pour que le nom verdict_method dans run() soit correctement défini

        :param details: dict - correspond à la réponse de fetch_data_from_source() pour VirusTotal
        :return: tuple (verdict, report_url, file_type) à utiliser dans les actions déclenchées sur
        TheHive par l'analyzer
        '''

        data = details.get("data")
        attributes = data.get("attributes")
        stats = attributes.get("last_analysis_stats")
        file_type = attributes.get("type_tag")

        if not data:
            self.error("Pas de champ 'data' dans la réponse de VirusTotal")
        elif not attributes:
            self.error("Le champ 'data' n'a pas de champ 'attributes' dans la réponse de VirusTotal")
        elif not stats:
            self.error("Le champ 'attributes' n'a pas de champ 'last_analysis_stats' dans la réponse de VirusTotal")

        else:
            if stats.get("malicious", 0) > 0:
                verdict = "Malicious"

            elif stats.get("suspicious", 0) > 0:
                verdict = "Suspicious"

            else:
                verdict = "Non-Malicious"

            report_id = details["data"]["id"]
            report_url = f"https://www.virustotal.com/gui/file/{report_id}"

            return verdict, report_url, file_type

    def verdict_OTX_Alienvault(self, details):
        '''
        Analyse les résultats de la requête à l'API de OTX_Alienvault

        Vérifie la présence des champs pertinents dans la réponse d'OTX et les utilise pour enrichir
        le hash analysé

        Adaptable pour d'autres sources si self.sources est modifié (ajout ou suppression de sources)
        Attention : bien utiliser le même nom que la clé correspondante dans self.sources dans le nom
        de la fonction pour que le nom verdict_method dans run() soit correctement défini

        :param details: dict - correspond au résultat de fetch_data_from_source() pour OTX
        :return: tuple (verdict, report_url, file_type) à utiliser dans les actions déclenchées sur
        TheHive par l'analyzer
        '''

        pulse_info = details.get("pulse_info")
        if not pulse_info:
            self.error("Pas de champ 'pulse_info' dans la réponse d'OTX")

        else:
            if pulse_info.get("count", 0) > 1:
                verdict = "Malicious"

            elif pulse_info.get("count", 0) == 1:
                verdict = "Suspicious"

            else:
                verdict = "Non-Malicious"

            report_id = details.get("indicator", 0)
            report_url = f"https://otx.alienvault.com/indicator/file/{report_id}"
            file_type = ""  # vide car sous absent dans les rapports OTX

            return verdict, report_url, file_type
    def summary(self, raw):
        '''
        Présente les verdicts des différentes sources de manière visuelle

        Construit une taxonomie par source avec un code couleur ('level') correspondant au verdict,
        le nom de l'analyzer ('namespace'), la source ('source') et le verdict ('verdict')
        Tient compte de la structure du dict results construit dans la fonction run() : ignore la Clé
        "Actions" qui concerne la fonction operations()

        :param raw: dict construit par run()
        :return: dict où la valeur est la liste des taxonomies construites
        '''

        taxonomies = []
        namespace = "HashAnalyzer"

        for source, result in raw.items():
            if source == "Actions":
                continue

            verdict_key = f"Verdict {source}"
            if verdict_key in result:
                verdict = result[verdict_key]
            else:
                self.error(f"Verdict manquant pour '{source}'")

            if verdict == "Malicious":
                level = "malicious"
            elif verdict == "Suspicious":
                level = "suspicious"
            else:
                level = "safe"

            taxonomies.append(self.build_taxonomy(level, namespace, source, verdict))

        return {"taxonomies": taxonomies}

    def artifacts(self, raw):
        '''
        Réécrit la méthode artifacts() car cet analyzer ne génère pas de nouvel observable
        '''

        return []

    # Les 3 fonctions suivantes gèrent les actions que l'analyzer peut être amené à réaliser sur
    # TheHive en fonction des résultats de l'analyse. Elles permettent d'éviter les erreurs dans
    # l'utilisation de self.build_operation() selon les cas tout en étant explicite sur les actions
    # réalisées

    def create_task(self, task):
        return self.build_operation("CreateTask", **task)

    def add_artifact_tag(self, label):
        return self.build_operation("AddTagToArtifact", tag=label)

    def add_case_tag(self, label):
        return self.build_operation("AddTagToCase", tag=label)

    def operations(self, raw):
        '''
        Construit la liste des opérations (actions) réalisées sur TheHive par l'analyzer

        Les actions sont effectivement réalisées dans run(), regroupées en une liste elle-même
        incluse dans les resultats sous la clé "Actions"

        :param raw: dict - en pratique sera le dictionnaire results défini dans run()
        :return: list operations
        '''

        operations = []

        if raw.get("Actions"):
            operations = operations + raw["Actions"]

        return operations

    def run(self):
        '''
        Utilise les fonctions antérieures pour enrichir l'observable et le case si pertinent

        Récupère le hash à analyser ;
        Regroupe les données fournies par les sources de self.sources sur le hash dans le dict results ;
        Utilise les fonctions verdict_{source} pour qualifier le hash ;
        Crée les tâches et tags (pour l'observable et le case) pertinents si le fichier est malicieux ou
        suspect ;
        Ajoute la liste des actions réalisées dans TheHive au dict results

        "results" aura la forme générale :
        {
            "Source": {
                fetch_data_from_source(Source, hash),                    (--> est un dict)
                "Verdict Source": "string"
                "Report Source": "string"
                "Country Source": "string"
            },
            "Actions": list [action 1, action 2...]
        }

        :return: application de la méthode report() de la classe Analyzer au dict results
        '''

        hash_value = self.get_data()
        results = {}
        malicious_sources = []
        suspicious_sources = []
        ops_done = []

        for source in self.sources.keys():
            data = self.fetch_data_from_source(source, hash_value)
            results[source] = {"data": data}

            try:
                verdict_method = getattr(self, f"verdict_{source}")
                verdict, report_url, file_type = verdict_method(data)
                results[source][f"Verdict {source}"] = verdict
                results[source][f"Report {source}"] = report_url
                results[source][f"File type {source}"] = file_type

                if verdict == "Malicious":
                    malicious_sources.append(source)

                if verdict == "Suspicious":
                    suspicious_sources.append(source)

            except AttributeError:
                self.error(f"Pas de méthode de verdict trouvée pour '{source}'")

        if malicious_sources or suspicious_sources:
            vt_report = results["VirusTotal"].get("Report VirusTotal")
            otx_report = results["OTX_Alienvault"].get("Report OTX_Alienvault")

            new_task_1 = {
                "title": "Analyze du fonctionnement du fichier malveillant",
                "description": "Voir aussi les rapports :\n\n"
                f"VirusTotal : {vt_report}\n\n"
                f"OTX-Alienvault : {otx_report}",
                "status": "Waiting"
            }
            task_op_1 = self.create_task(new_task_1)
            ops_done.append(task_op_1)

            new_task_2 = {
                "title": "Suppression du fichier malveillant",
                "description": f"Vérifier sur l'ensemble du SI Echelon - hash : {hash_value}",
                "status": "Waiting"
            }
            task_op_2 = self.create_task(new_task_2)
            ops_done.append(task_op_2)

            new_task_3 = {
                "title": "Mise à jour des outils de sécurité pour détecter le programme malveillant",
                "description": f"EDR et autres outils de détection - hash : {hash_value}",
                "status": "Waiting"
            }
            task_op_3 = self.create_task(new_task_3)
            ops_done.append(task_op_3)

        art_tag = results["VirusTotal"]["File type VirusTotal"]
        tag_op = self.add_artifact_tag(art_tag)
        ops_done.append(tag_op)

        art_tag_2 = "cortex analyzer"
        tag_op_2 = self.add_artifact_tag(art_tag_2)
        ops_done.append(tag_op_2)

        case_tag = "hash analyzer"
        case_tag_op = self.add_case_tag(case_tag)
        ops_done.append(case_tag_op)

        results["Actions"] = ops_done
        self.report(results)


if __name__ == "__main__":
    HashAnalyzer().run()
```

HostAnalyzer.py

```python
#!/usr/bin/env python3
# encoding: utf-8

# Date : novembre 2024
# Version : 1.0
# Auteur : Nicolas Clerbout

# Description : Analyzer Cortex pour enrichir des observables hostname sur TheHive en utilisant l'annuaire
# OpenLDAP
# Autres informations (licence, etc) et configuration dans le fichier HostAnalyzer.json

from cortexutils.analyzer import Analyzer
import ldap3
import re
from ldap3 import Server, Connection, SIMPLE, SYNC, SUBTREE, ALL


class HostAnalyzer(Analyzer):
    def __init__(self):
        '''
        Récupère les données de configuration et teste la connexion à l'annuaire OpenLDAP
        '''

        Analyzer.__init__(self)

        self.ldap_address = self.get_param("config.ldap_address", None, "Missing LDAP address")
        self.ldap_port = self.get_param("config.ldap_port", None, "Missing LDAP port")
        self.ldap_port = int(self.ldap_port)

        self.username = self.get_param("config.username", None, "Missing username")
        self.password = self.get_param("config.password", None, "Missing password")
        self.base_dn = self.get_param("config.base_DN", None, "Missing base DN")
        self.user_dn = f"cn={self.username},{self.base_dn}"

        try:
            s = Server(self.ldap_address, port=self.ldap_port, get_info=ALL, use_ssl=False)
            self.connection = Connection(s, auto_bind=True, client_strategy=SYNC, user=self.user_dn,
                                         password=self.password, authentication=SIMPLE, check_names=True)

        except Exception as e:
            self.error(f"Erreur de connexion LDAP : {e}")

    def get_host_infos(self, host_id):
        '''
        Construit une liste d'informations sur l'hôte qui seront ajoutées comme tags à l'observable

        Fait une recherche filtrée par uid dans l'annuaire pour récupérer certains attributs.
        Possibilité d'ajouter des attributs selon les besoins sans besoin de modifier run()

        :param host_id: string
        :return: list - liste des attributs sélectionnés dans la recherche filtrée via "attributes="
        '''

        host_infos = []
        filter_hostid = f"(uid={host_id})"
        self.connection.search(self.base_dn, filter_hostid, SUBTREE, attributes=["description"])
        responses = self.connection.response

        if responses:
            for response in responses:
                dict_response = response.get("attributes", None)
                info = {}
                if dict_response:
                    for att in dict_response.keys():
                        info[att] = dict_response[att]
                    host_infos.append(info)

        self.connection.unbind()

        return host_infos

    def get_uid(self, name):
        """Recherche et retourne l'UID correspondant au nom donné, en supposant un unique résultat."""

        filter_cn = f"(cn=*.{name})"
        self.connection.search(self.base_dn, filter_cn, SUBTREE, attributes=["uid"])

        # Vérifie s'il y a un résultat et retourne directement l'UID sans créer de liste
        if self.connection.response and "uid" in self.connection.response[0]["attributes"]:
            return self.connection.response[0]["attributes"]["uid"][0]
        return None

    def find_user(self, host):
        '''
        Vérifie si le nom d'hôte correspond à un poste de travail et récupère le nom d'utilisateur

        S'appuie sur la convention de nommage des machines dans l'annuaire OpenLDAP : si le nom est
        WKS-ECH-NAME, utilise NAME comme paramètre pour trouver l'utilisateur correspondant grâce à
        get_uid().

        :param host: string - sera l'observable (nom d'hôte) analysé
        :return: string - nom d'utilisateur à ajouter comme observable au case avec create_observable() ou
        message d'erreur si la machine n'est pas un poste de travail
        '''

        try:
            match = re.match(r"^WKS-ECH-(\w+)$", host)
            if match:
                name = match.group(1)
                user_id = self.get_uid(name)
                return user_id

            else:
                return "Not a workstation"  # la machine n'est pas un poste de travail

        except Exception as e:
            self.unexpectedError(e)
            return None

    def summary(self, raw):
        '''
        Présente le résultat de find_user() de manière visuelle

        S'appuie sur le résultat de find_user() quand appelé dans run() : vérifie si l'info "User" fait
        partie du rapport de l'analyzer et retourne une valeur (value) selon les cas

        :param raw: dict - sera le dictionnaire results défini dans run()
        :return: dict - où la valeur de "taxonomies" sera une liste à un seul élément
        '''

        taxonomies = []
        level = "info"
        namespace = "(Host)"
        predicate = "Assigned to"
        value = "?"

        if raw["User"]:
            value = raw["User"]

        taxonomies.append(self.build_taxonomy(level, namespace, predicate, value))

        return {"taxonomies": taxonomies}

    def artifacts(self, raw):
        '''
        Construit une liste d'observables à ajouter au case

        Vérifie si find_user() a retourné un résultat quand appelée dans run(). Si oui, et si la machine est
        un poste de travail, ajoute le nom d'utilisateur comme observable de type username importable dans
        TheHive

        N.B. : cet observable n'est pas automatiquement importé dans TheHive à partir de la liste artifacts.
        C'est pourquoi on utilise create_observable() dans run() pour ajouter l'observable sans besoin
        d'intervention humaine

        :param raw: dict - sera le dictionnaire results défini dans run()
        :return: list - avec le nom d'utilisateur comme seul élément, le cas échéant
        '''

        artifacts = []

        if raw.get("User"):
            if raw["User"] != "Not a workstation":
                artifact_type = "username"
                name = raw["User"]
                tags = []
                artifacts.append(self.build_artifact(artifact_type, name, tags=tags))

        return artifacts

    # Les 4 fonctions suivantes gèrent les actions que l'analyzer peut être amené à réaliser sur
    # TheHive en fonction des résultats de l'analyse. Elles permettent d'éviter les erreurs dans
    # l'utilisation de self.build_operation() selon les cas tout en étant explicite sur les actions
    # réalisées

    def create_observable(self, observable):
        return self.build_operation("AddArtifactToCase", **observable)

    def create_task(self, task):
        return self.build_operation("CreateTask", **task)

    def add_artifact_tag(self, label):
        return self.build_operation("AddTagToArtifact", tag=label)

    def add_case_tag(self, label):
        return self.build_operation("AddTagToCase", tag=label)

    def operations(self, raw):
        '''
        Construit une liste d'opérations à ajouter au rapport

        Utilise les actions effectuées et listées dans run() sous la clé "Actions"

        :param raw: dict - sera le dictionnaire results défini dans run()
        :return: list - liste des actions effectuées dans TheHive durant l'application de run()
        '''

        operations = []

        if raw.get("Actions"):
            operations = operations + raw["Actions"]

        return operations

    def run(self):
        '''
        Analyse et enrichit l'observable de type hostname

        Récupère le nom d'hôte, applique les fonctions find_user() et get_host_infos() pour l'enrichir
        Construit le dictionnaire results qui a la forme suivante :
        {
          "User": "<nom_d_utilisateur>",      -- si la machine est un poste de travail
          "HostDetails": list,                -- attributs obtenus avec get_host_infos()
          "Actions": list                     -- actions d'enrichissement réalisées dans TheHive
        }

        :return: application de self.report() au dictionnaire results
        '''

        host = self.get_data()
        results = {}
        ops_done = []

        if self.data_type == "hostname":
            user_id = self.find_user(host)
            if user_id is None:
                return

            elif user_id != "Not a workstation":
                results["User"] = user_id

                # Si un utilisateur est trouvé, créé un observable de type username
```

```python
                new_observable = {
                    "dataType": "username",
                    "data": user_id,
                    "tags": [],
                    "message": ""
                }
                observable_op = self.create_observable(new_observable)
                ops_done.append(observable_op)

            try:
                host_infos = self.get_host_infos(host)
                if not host_infos:
                    self.error(f"Pas d'information trouvée pour {host}")
                    return

                results["HostDetails"] = host_infos

                for i in range(len(host_infos)):
                    if not isinstance(host_infos[i], str):
                        host_infos[i] = str(host_infos[i])  # assure que les infos sont des strings

                    # ajoute les infos listées dans host_infos comme tags à l'observable
                    new_tag = host_infos[i]
                    tag_op = self.add_artifact_tag(new_tag)
                    ops_done.append(tag_op)

            except Exception as e:
                self.unexpectedError(e)

            # Autres actions réalisées sur l'observable et le case
            art_tag = "cortex analyzer"
            tag_op_2 = self.add_artifact_tag(art_tag)
            ops_done.append(tag_op_2)

            task = {
                "title": f"Analyse du trafic réseau sur {host}",
                "description": "Consulter logs trafic réseau, faire une capture de trafic si nécessaire",
                "status": "Waiting"
            }
            task_op = self.create_task(task)
            ops_done.append(task_op)

            case_tag = "hostname analyzer"
            case_tag_op = self.add_case_tag(case_tag)
            ops_done.append(case_tag_op)

            results["Actions"] = ops_done
            self.report(results)

        else:
            self.notSupported()


if __name__ == "__main__":
    HostAnalyzer().run()
```

UserAnalyzer.py

```python
#!/usr/bin/env python3
# encoding: utf-8

# Date : novembre 2024
# Version : 1.0
# Auteur : Nicolas Clerbout

# Description : Analyzer Cortex pour enrichir des observables username sur TheHive en utilisant l'annuaire
# OpenLDAP
# Autres informations (licence, etc) et configuration dans le fichier UserAnalyzer.json

import re
from cortexutils.analyzer import Analyzer
import ldap3
from ldap3 import Server, Connection, SIMPLE, SYNC, SUBTREE, ALL


class UserAnalyzer(Analyzer):
    def __init__(self):
        '''Récupère les données de configuration et teste la connexion à l'annuaire OpenLDAP'''

        Analyzer.__init__(self)

        self.ldap_address = self.get_param("config.ldap_address", None, "Missing LDAP address")
        self.ldap_port = self.get_param("config.ldap_port", None, "Missing LDAP port")
        self.ldap_port = int(self.ldap_port)

        self.username = self.get_param("config.username", None, "Missing username")
        self.password = self.get_param("config.password", None, "Missing password")
        self.base_dn = self.get_param("config.base_DN", None, "Missing base DN")
        self.user_dn = f"cn={self.username},{self.base_dn}"

        try:
            s = Server(self.ldap_address, port=self.ldap_port, get_info=ALL, use_ssl=False)
            self.connection = Connection(s, auto_bind=True, client_strategy=SYNC, user=self.user_dn,
                                         password=self.password, authentication=SIMPLE, check_names=True)

        except Exception as e:
            self.error(f"Erreur de connexion LDAP : {e}")

    def get_details(self, user_id):
        '''
        Construit une liste d'informations sur l'utilisateur qui seront ajoutées comme tags à l'observable

        Fait une recherche filtrée par uid dans l'annuaire pour récupérer certains attributs.
        Possibilité d'ajouter ou supprimer des attributs selon les besoins sans besoin de modifier run()

        :param user_id: string
        :return: list - liste des attributs sélectionnés dans la recherche filtrée via "attributes="
        '''

        filter_uid = f"(uid={user_id})"
        self.connection.search(self.base_dn, filter_uid, SUBTREE, attributes=["mail", "description"])

        if self.connection.response:
            details = []
            for response in self.connection.response:
                for attribute, value in response["attributes"].items():
                    details.append(f"{attribute} : {value[0]}")

            return details

        else:
            return []

    def get_group(self, user_id):
        """Recherche et retourne les noms des groupes associés à un user_id donné."""

        filter_member_uid = f"(memberUid={user_id})"
        self.connection.search(self.base_dn, filter_member_uid, SUBTREE, attributes=["cn"])

        groups = []
        for response in self.connection.response:
            if "cn" in response["attributes"]:
                groups.append(response["attributes"]["cn"][0])
        if groups:
            return " ; ".join(groups)  # rassemble les différents groupes en une string (pour ajout comme tag)
        else:
            return []

    def summary(self, raw):
        '''
        Fournit une indication visuelle d'analyse effectuée

        Utilise la fonction self.build_taxonomy(). Indique si l'observable a été enrichi avec des tags dans
        run()

        :param raw: dict - sera le dictionnaire results défini dans run()
        :return: dict - avec une seule entrée où la valeur est la taxonomie construite avec
        self.build_taxonomy()
        '''

        taxonomies = []
        level = "info"
        namespace = "(User)"
        predicate = "Details"
        value = "?"

        if raw.get("Details"):
            value = "cf tags"

        taxonomies.append(self.build_taxonomy(level, namespace, predicate, value))
        return {"taxonomies": taxonomies}

    def artifacts(self, raw):
        '''
        Réécrit la méthode artifacts() de la classe Analyzer car cet analyzer ne génère pas de nouvel
        observable
        '''

        artifacts = []

        return artifacts

    # Les 3 fonctions suivantes gèrent les actions que l'analyzer peut être amené à réaliser sur
    # TheHive en fonction des résultats de l'analyse. Elles permettent d'éviter les erreurs dans
    # l'utilisation de self.build_operation() selon les cas tout en étant explicite sur les actions
    # réalisées

    def create_task(self, task):
        return self.build_operation("CreateTask", **task)

    def add_artifact_tag(self, label):
        return self.build_operation("AddTagToArtifact", tag=label)

    def add_case_tag(self, label):
        return self.build_operation("AddTagToCase", tag=label)

    def operations(self, raw):
        '''
        Construit une liste d'opérations à ajouter au rapport

        Utilise les actions effectuées et listées dans run() sous la clé "Actions"

        :param raw: dict - sera le dictionnaire results défini dans run()
        :return: list - liste des actions effectuées dans TheHive durant l'application de run()
        '''

        operations = []

        if raw.get("Actions"):
            operations = operations + raw["Actions"]

        return operations

    def run(self):
        '''
        Analyse et enrichit l'observable de type username

        Récupère le nom d'utilisateur, applique les fonctions get_details() et get_group() pour l'enrichir
        Construit le dictionnaire results qui a la forme suivante :
        {
          "User": "<nom_d_utilisateur>",      -- observable analysé
          "Details": list,                    -- attributs obtenus avec get_details()
          "Actions": list                     -- actions réalisées dans TheHive
        }

        :return: application de self.report() au dictionnaire results
        '''

        user_id = self.get_data()
        results = {}
        ops_done = []

        if self.data_type == "username":
            try:
                details = self.get_details(user_id)
                groups = self.get_group(user_id)

                # Construit la liste des tags et les ajoute à l'observable
                if groups:
                    artifact_tags = details + [f"groups: {groups}"] + ["cortex analyzer"]
                else:
                    artifact_tags = details + ["groups: Aucun"] + ["cortex analyzer"]

                for i in range(len(artifact_tags)):
                    if not isinstance(artifact_tags[i], str):
                        artifact_tags[i] = str(artifact_tags[i])

                    newtag = artifact_tags[i]
                    tag_op = self.add_artifact_tag(newtag)
                    ops_done.append(tag_op)

                # Autres actions d'enrichissement + mise à jour du case
                new_task = {
                    "title": f"Vérifier les logs de connexions de {user_id}",
                    "description": "Vérifier si c'est bien l'utilisateur connecté à la machine signalée lors du "
                                   "comportement suspect",
                    "status": "Waiting"
                }
                task_op = self.create_task(new_task)
                ops_done.append(task_op)

                case_tag = "username analyzer"
                case_tag_op = self.add_case_tag(case_tag)
                ops_done.append(case_tag_op)

                results = {
                    "User": user_id,
                    "Details": artifact_tags,
                    "Actions": ops_done
                }

                self.report(results)

            except Exception as e:
                self.unexpectedError(e)

        else:
            self.notSupported()


if __name__ == "__main__":
    UserAnalyzer().run()
```