Université Lumière Lyon 2

Projet d'application
Livrable 2 : EaSIEM

Tom Areal et Thomas Lecomte
Automne 2025

Sommaire

Mise en situation

Dans le cadre de la création d'un **gestionnaire de journaux de sécurité** (Security Log Manager), nous avions annoncé dans notre premier livrable que l'application devait **1)** collecter des logs, **2)** harmoniser leur format, **3)** analyser leur contenu, **4)** produire des rapports d'événements de sécurité, **5)** archiver les rapports et les logs utilisés.

Pour répondre à ces exigences, nous avons décidé de considérer que le gestionnaire de journaux serait implémenté dans un système d'information composé d'une machine Windows et d'une machine Linux. L'objectif est de mettre en perspective la différence native entre les logs de ces deux OS. Quatre types de logs ont finalement été retenus pour notre gestionnaire : deux logs système (*Linux syslog* et *Windows System log*) et deux logs de sécurité (*Linux secure* et *Windows security log*). Nous avons fait ce choix afin d'avoir des formats et des sources variés d'information. Cela nous laisse également plus de possibilités dans le type d'attaque à détecter lors de notre scénarisation finale.

Rôles, relations et sémantiques des classes

Nous avons identifié **dix (10) classes principales** à mettre en place pour que notre gestionnaire de journaux fonctionne convenablement selon notre cahier des charges:

1. *Log_Collector*. Cette classe est en quelque sorte la « porte d'entrée » de notre gestionnaire. Log_Collector **importe** les fichiers des logs et les **parcourt**, afin d'**identifier le type de log** concerné[1] et les balises utilisées. Log_Collector **découpe ensuite le contenu** des fichiers en fonction de ces balises et **instancie un objet** avec. Le contenu des balises sert ainsi à remplir les attributs spécifiques de chaque classe liée aux logs. Afin de mener à bien cette mission, Log_Collector utilise la bibliothèque **pugixml**, qui permet de gérer les fichiers en format .xml (format retenu pour nos logs test).
    - La classe Log_Collector (1) est en **relation d'agrégation** avec la classe Log (1..*). On ne peut pas avoir de Log sans Log_Collector, mais si Log_Collector est détruit, les objets instanciés dans Log persisteront.
2. *Log* (**classe mère**). Cette classe contient les objets instanciés via la méthode de Log_Collector. Log regroupe notamment les attributs communs à chaque type de logs présents dans notre gestionnaire. C'est à partir des objets instanciés dans la classe mère Log (et dans ses sous-classes à fortiori) que l'on procédera à l'analyse et à la création d'événements de sécurité.

[1] Linux syslog, Linux secure, Windows security log, Windows system log

- La classe Log est en **relation d'héritage** avec les **quatre (4) sous-classes** Windows_System_Log, Linux_System_Log, Windows_Security_Log et Linux_Secure_Log.
- La classe Log (*) est en relation avec la classe Security_Event (1).
- La classe Log (*) est en relation avec la classe Event_Rapport (1).
- La classe Log (1..*) est en relation avec la classe Event_Log_Filter (1..*).

3. *Windows_System_Log* (**classe fille de Log**). Cette classe contient les attributs spécifiques au *Windows system log* et hérite des attributs de Log.
4. *Linux_System_Log* (**classe fille de Log**). Cette classe contient les attributs spécifiques au *Linux syslog* et hérite des attributs de Log.
5. *Windows_Security_Log* (**classe fille de Log**). Cette classe contient les attributs spécifiques au *Windows security log* et hérite des attributs de Log.
6. *Linux_Security_Log* (**classe fille de Log**). Cette classe contient les attributs spécifiques au *Linux secure* et hérite des attributs de Log.
7. *Security_Event*. Cette classe permet d'**analyser les attributs des entrées dans Log** et de **faire des corrélations** pour **identifier des événements de sécurité**. Pour chaque type d'attaque que l'on souhaite reconnaitre, on élabore un **pattern** (ensemble de critères précis) et on l'implémente dans la méthode d'analyse de la classe. Les événements de sécurité ainsi détectés sont instanciés comme des objets dans Security_Event.
    - La classe Security_Event (1) est en relation avec la classe Log (1..*).
    - La classe Security_Event (*) est en relation avec la classe Event_Rapport (1).
    - La classe Security_Event (1) est en relation avec la classe Archives (1).
8. *Event_Rapport* (**classe d'association**). Cette classe permet d'**analyser des événements** de sécurité et de **faire des corrélations** pour **identifier les événements pouvant appartenir à une même séquence d'attaque**. Le rapport permet de déterminer si les événements sont liés et s'ils font sens. Comme pour la classe Security_Event, on établit un **pattern** de reconnaissance et on l'implémente dans la méthode d'analyse de la classe. Les rapports ainsi produits sont instanciés comme des objets dans Event_Rapport. On veut que le rapport soit une **synthèse « clé en main »** permettant d'**évaluer la priorité/gravité** de la séquence d'attaque qui a eu lieu. Les rapports finaux sont exportables en format .txt à partir de la classe Event_Rapport.
    - La classe Event_Rapport (1) est en relation avec la classe Security_Event (*).
    - La classe Event_Rapport (1) est en relation avec la classe Log (*).
    - La classe Event_Rapport (1) est en relation avec la classe Archives (1).

9. *Archives*. Cette classe permet de sauvegarder, pour une durée déterminée, les objets instanciés dans les classes Log, Security_Event et Event_Rapport. C'est aussi à partir de cette classe qu'il est possible d'opérer une recherche pour retrouver un objet ayant été instancié dans l'une de ces trois (3) classes .
    - La classe Archives (1) est en relation avec la classe Security_Event (1).
    - La classe Archives (1) est en relation avec la classe Security_Event (1).
    - La classe Archives (1) est en relation avec la classe Security_Event (1).

A ces neuf (9) classes principales **s'ajoute également une classe d'association Exeption** qui a été implémentée, à titre d'exemple, dans le présent diagramme de classes. Cette classe, ici positionnée entre Log_Collector et Log, pourrait être dupliquée un peu partout entre les classes de notre diagramme, afin de gérer les exceptions de façon sécuritaire.

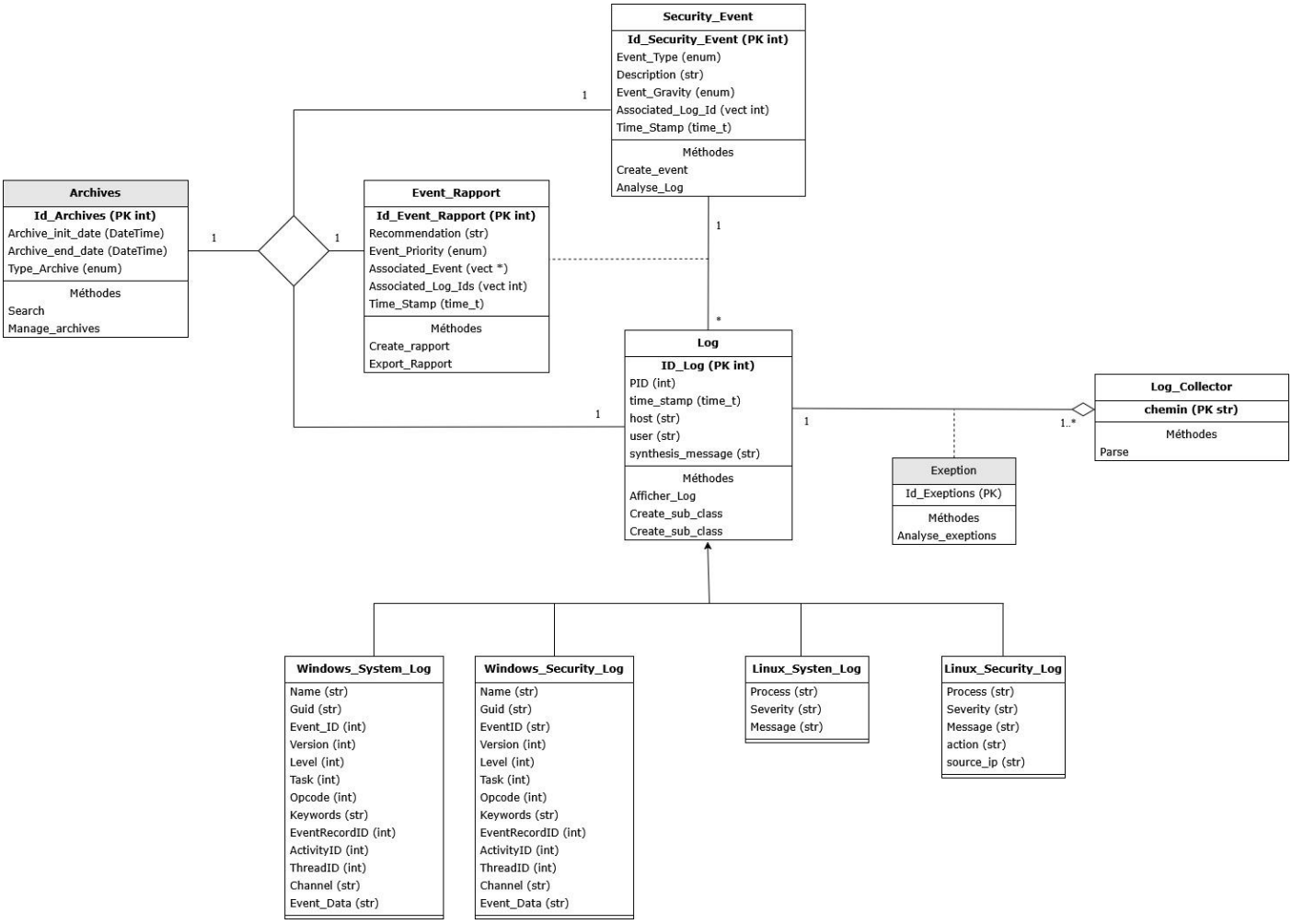
10. *Exeption* (**classe d'association**). On peut imaginer que, positionnée entre Log_Collector et Log, la classe Exeption pourrait principalement servir dans deux cas de figures : lorsqu'un attribut de Log n'a pas de valeur à recevoir (NULL) ou lorsqu'une valeur isolée par Log_Collector n'a pas d'attribut établi dans Log.

Remarque concernant l'héritage dans la conception des classes

Lors de la conception de notre classe Log et ses sous-classes, nous avons exploré plusieurs possibilités pour traduire la relation d'héritage. Le fait est que les classes héritières, en fonction de l'OS auxquels elles sont liées, ont beaucoup de leurs attributs en commun. Nous avons donc dû choisir entre le fait de 1) garder toutes les sous-classes séparées mais avoir des redondances dans le diagramme de classes, 2) réunir les classes par OS et initialiser les attributs distinctifs à NULL lorsque nous ne pouvons pas les remplir.

Nous avons finalement fait le choix de garder les sous-classes séparées, afin d'avoir un diagramme au contenu plus explicite, même si ce choix pourrait être discuté.

Diagramme de classe



Remarque concernant le diagramme de classe

* Dans le présent diagramme de classes, nous précisons les types de données correspondant à chacun des attributs de nos classes (en nous appuyant notamment sur les types utilisés nativement dans les logs Windows et Linux). Toutefois, dans notre code, nous avons pris la liberté de n'implanter ces types que pour les attributs indispensables à nos tests de fonctionnement. Les attributs présents mais inutilisés dans nos fonctions ont simplement reçus un type string standard. Non seulement ce choix ne change rien au déroulé de notre preuve de concept, mais en plus il facilite la fonction d'affichage si jamais on veut exposer l'entièreté du contenu de nos objets dans le terminal.

** Dans la classe Log_Collector, nous indiquons que l'attribut « chemin » (str) est utilisé comme clé primaire. Nous savons que ce choix n'est pas orthodoxe, mais il s'avère que le chemin vers les fichiers

.xml utilisés par le Log_Collector est toujours un string unique. Nous ne voyions donc pas la pertinence d'ajouter un attribut Id_Log_Collector à part entière.

*** Le diagramme de classes qui est illustré ici ne reflète pas exactement notre production finale. En effet, en raison de contraintes de temps, les classes Archives et Exeption ont bien été codées en .hpp, mais elles n'ont pas pu recevoir leurs .cpp affiliés. Ces classes sont donc présentes, nous trouvons pertinent de signaler leur existence, mais elles n'interagissent pas avec le reste de notre code. Bien que ce soient des « coquilles vides », nous trouvions dommage de les occulter tellement elles semblent indispensables à tout *logs manager* qui se respecte.

**** Afin de simplifier la compréhension des méthodes utilisées dans les classes, nous avons pris la liberté de donner des noms suggestifs à nos méthodes dans ce diagramme. Attention, ces noms ne sont pas ceux présents dans nos algorithmes, ils servent seulement à se figurer les fonctionnalités principales associées à chaque classe.

Difficultés techniques identifiées

Deux difficultés principales sont identifiables lorsqu'on analyse notre système de gestion des journaux de sécurité. La première difficulté survient lorsqu'on doit **transformer nos logs bruts** (format .xml) **en un ensemble d'objets structurés et instanciables** dans notre classe Log (et dans ses classes héritières). Il faut en effet s'assurer que le *parsing* s'adapte à chacun de nos types de logs et que le découpage harmonise correctement le format des attributs transversaux à tous les logs. Un autre point important à considérer lors de cette étape concerne l'implémentation d'un tri chronologique des objets à partir du *timestamp* de création indiqué dans les logs d'origine. Il faut en effet que l'ensemble des entrées dans les logs soit ordonné dans le temps si on veut identifier des attaques touchant notre système d'information.

La seconde difficulté concerne quant à elle l'**analyse des objets présents dans les logs**, lorsque l'on cherche à faire des corrélations et que l'on veut en déduire des événements de sécurité. Il faut alors établir des critères cohérents et réalistes pour déterminer le type d'attaque qui constituera un événement de sécurité. Il est également nécessaire de comparer les logs ensemble et d'encapsuler les données de multiples logs dans un seul et même événement. En maitrisant la logique derrière la création des événements, il devient beaucoup plus facile d'implémenter aussi la création des rapports qui compilent des événements aux caractéristiques similaires.

Scénarisation proposée

Afin de mettre à l'épreuve notre application et de fournir une **preuve de concept** pour chacune des difficultés susmentionnées, nous avons prévu une scénarisation complète. Cette scénarisation se déroule en **deux temps** :

1. Nous proposons d'abord d'**utiliser deux (2) fichiers .xml contenant des entrées de logs « sains »**, afin de tester la fonction de *parsing* de la classe Log_Collector. L'objectif est de vérifier que nos objets dans la classe Log (et dans ses héritières) sont correctement instanciés. A ce stade, les fichiers .xml utilisés sont Linux_Syslog.xml et Windows_System_Log.xml. Ils ne contiennent aucun indice alarmant concernant une attaque.
2. Nous proposons ensuite de **rajouter deux (2) autres fichiers .xml contenant des entrées de logs « suspectes »**, afin de tester la fonction de détection d'attaque de la classe Security_Event. Les fichiers .xml utilisés pour le test de détection sont Linux_Secure_Suspect.xml et Windows_Security_Log_Suspect.xml. Ils contiennent notamment des données liées à des tentatives de connexion échouées, réalisées dans un délai court (moins de 30 secondes). Ces activités doivent amener notre algorithme de détection à identifier deux (2) attaques par brute force et à les indexer comme des événements de sécurité.

   Dans un second temps, le système va établir une corrélation entre ces deux (2) attaques en raison de leur nature (brute force) et du fait qu'elles sont rapprochées dans le temps. La compilation de ces événements permet de produire un rapport de sécurité concernant l'ensemble de notre système d'information. Ce rapport comprend des recommandations et établit une priorité d'action.

Réussites et limites

Nous savions initialement que la création de notre application serait contrainte par un certain manque de temps. Malgré cela, il est possible que nous ayons eu du mal à restreindre nos ambitions face à la stimulation générée par ce projet... Bien que nous ayons réussi à relever les deux difficultés techniques principales annoncées dans ce rapport, nous n'avons pas eu le temps de pleinement coder les classes Archives et Exeption (ainsi que leurs fonctionnalités associées). Nous convenons aussi que la production du rapport final, qui corrèle ensemble des évènements de sécurité, reste assez sommaire en comparaison de ce qui nous aurions pu proposer avec un peu plus de temps.

Au final, nous sommes tout de même parvenus à répondre au cahier des charges imposé, les fonctions de *parsing* et d'analyse de log sont fonctionnelles et nous sommes en capacité de vous proposer une scénarisation complète pour notre gestionnaire.

Annexe : Satisfaction du cahier des charges et des exigences minimales

Implémentation d'au moins 4 classes en plus d'un programme main contenant la création des objets et l'appel des méthodes des classes :

Notre système de gestion des journaux de sécurité repose sur dix (10) classes (dont huit (8) utilisées dans notre démonstration) et sur le programme *main*. Précisons toutefois que, compte tenu des mécaniques à l'œuvre dans notre application, la création d'objets se fait dans les fichiers .cpp de nos classes et non directement dans le *main*. L'appel des méthodes se fait quant à lui dans le *main* et dans les classes, dépendamment des besoins.

Les classes doivent comporter des attributs et des méthodes avec des spécificateurs d'accès public, privé et protégé, justifiés.

Les trois spécificateurs d'accès public, privé et protégé sont bien implémentés dans l'ensemble de notre application. Les *screens* ci-joints, pris dans la classe mère Log et dans la sous-classe Windows_System_Log, illustrent bien la présence de ces spécificateurs.

```
class Log {
    protected:

        int ID_Log = -1; //attributs et fonctions que les classes héritées auront
        int PID = -1;
        time_t time_stamp = 0;
        std::string host = "";
        std::string user = "";
        std::string synthesis_message = "";

    public:

        Log(int id,
        const int& pid,
        const time_t& time_stamp,
        const std::string& h,
        const std::string& u,
        const std::string& synthesis_message);
```

```
class Windows_System_Log : public Log{
    private:

        std::string name = "";
        std::string guid = "";
        int event_ID = -1;
        std::string version = "";
        std::string level = "";
        std::string task = "";
        std::string opcode = "";
        std::string keywords = "";
        std::string eventRecordID = "";
        std::string activityID = "";
        std::string threadID = "";
        std::string channel = "";
        std::string eventData = "";
```

Dans notre cas (l'héritage et seulement l'héritage), la mention *protected* permet de garantir l'accès des attributs aux sous-classes de Log. Ces dernières peuvent alors utiliser les attributs et les modifier de façon sécuritaire, sans utiliser de *getters* ou de *setters*.

Les attributs en *private* sont quant à eux inaccessibles depuis l'extérieur des classes, ce qui évite les modifications indésirables. On utilise alors les *getters* et les *setters* pour les manipuler.

Enfin, on est aussi parfois contraint de mettre des attributs en *public* (même si c'est dangereux). C'est le cas quand l'utilisateur doit accéder aux attributs d'un constructeur pour instancier un objet dans une classe.

Pour chacune des classes, fournir un fichier entête (.hpp) déclarant les attributs et les prototypes, et un fichier source (.cpp) avec l'implémentation.

Le contenu de l'entièreté de nos classes est bien reparti entre nos fichiers .hpp et .cpp.

Dans le *main*, l'initialisation et la récupération des valeurs des attributs doivent se faire via getters et setters.

Comme on peut le voir dans ces *screens* issus de Log.hpp et de Log.cpp, nous utilisons constamment des getters et des setters pour accéder à nos attributs lorsqu'ils ne se trouvent pas dans les mêmes classes. Notons toutefois qu'en raison de la conceptualisation de notre système, nous n'initialisons et nous ne récupérons pas les valeurs d'attributs directement dans le *main*.

```cpp
time_t getTimeStamp() const;

void setSynthesisMessage(const std::string& message);
```

```cpp
time_t Log::getTimeStamp() const {
    return time_stamp;
}

void Log::setSynthesisMessage(const std::string& message) {
    synthesis_message = message;
}
```

L'application doit contenir des comparaisons entre objets à l'aide des opérateurs (operator==(), operator>(), operator<(), etc.).

La comparaison entre objets est bien implémentée via la fonction de tri des entrées dans la classe Log (et ses classes héritières). La fonction de tri repose principalement sur l'opérateur **operator<()** afin d'ordonner chronologiquement nos entrées de log, quelle que soit leur origine.

```cpp
bool operator<(const Log& other) const {
    return this->getTimeStamp() < other.getTimeStamp();
}

bool operator>(const Log& other) const {
    return this->getTimeStamp() > other.getTimeStamp();
}

bool operator==(const Log& other) const {
    return this->getTimeStamp() == other.getTimeStamp();
}
```

```cpp
std::vector<std::unique_ptr<Log>> trier_logs_croissant(
    std::vector<std::unique_ptr<Log>> logs
) {
    std::sort(
        logs.begin(),
        logs.end(),
        [](const std::unique_ptr<Log>& a, const std::unique_ptr<Log>& b) {
            return *a < *b;
        }
    );

    return std::move(logs); // move implicite
}

std::vector<std::unique_ptr<Log>> trier_logs_decroissant(
    std::vector<std::unique_ptr<Log>> logs
) {
    std::sort(
        logs.begin(),
        logs.end(),
        [](const std::unique_ptr<Log>& a, const std::unique_ptr<Log>& b) {
            return *a > *b;
        }
    );

    return std::move(logs); // move implicite
}
```

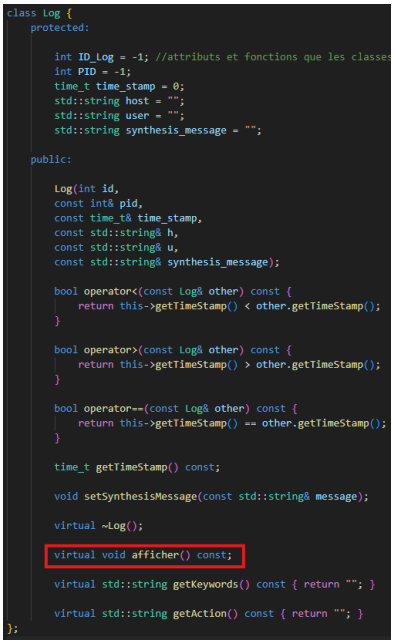
Ici, dans le main.cpp, on utilise des opérateurs par défaut afin de trier une liste de pointeurs vers des entrées de logs.

Au moins une relation d'association, d'agrégation ou de composition entre les classes, ainsi qu'un héritage.

Comme expliqué au début de ce rapport, une relation d'héritage est présente via la classe Log, une relation d'association est présente via la classe Event_Rapport, une relation d'agrégation est présente via la classe Log_Collector.

Présence d'au moins une méthode virtuelle (polymorphisme).

Comme on peut le voir dans Log.hpp, la méthode d'affichage nous fournit un parfait exemple de polymorphisme. Quand la méthode sera appelée, l'algorithme va regarder l'origine de cet appel et va adapter l'affichage en conséquence, en fonction des attributs de la classe concernée. Grace a *override*, les classes héritières de Log vont pouvoir recréer une méthode d'affichage sur mesure.



Gestion de la lecture/écriture depuis/sur des fichiers.

La gestion de la lecture, va essentiellement se retrouver dans Log_Collector.cpp, lorsqu'il s'agira de de lire les fichiers .xml.

Pour la gestion de l'écriture, c'est dans la classe Event_Rapport.cpp qu'il faut regarder, lorsque nous exportons les rapports de sécurité vers un fichier au format .txt.

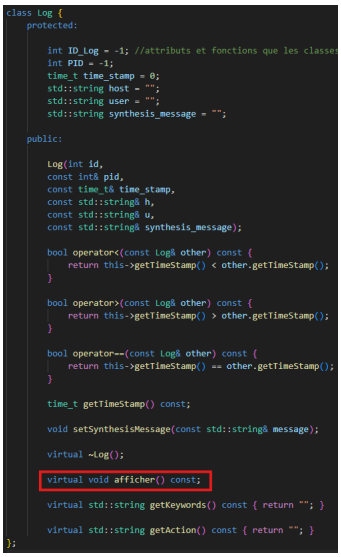
```cpp
std::vector<std::unique_ptr<Log>> Log_collector::importer()   {
    std::vector<std::unique_ptr<Log>> logs;
    pugi::xml_document xml;
    pugi::xml_parse_result parsing = xml.load_file(chemin.c_str());
    if (xml.child("Events").attribute("log_type")) {
        // l'attribut existe linux
        if (xml.child("Events").attribute("log_type").as_string() == std::string("secure")) {
            // linux sec
            pugi::xml_node Events = xml.child("Events");
            for (pugi::xml_node Event : Events.children("event")) {
                logs.push_back(std::make_unique<Linux_Security_Log>(
                    Log(Global_Id_Log++,
                    Event.child("pid").text().as_int(),
                    convertisseur_date(Event.child("timestamp").text().as_string()), //a modif
                    Event.child("hostname").text().as_string(),
                    Event.child("user").text().as_string(),
                    ""),

                    Event.child("process").text().as_string(),
                    Event.child("sevenity").text().as_string(),
                    Event.child("action").text().as_string(),
                    Event.child("source_ip").text().as_string(),
                    Event.child("message").text().as_string()
                ));
            }
            return logs;
```

Utilisation de pointeurs, méthodes const, transmissions par valeur et par référence, avec justification.

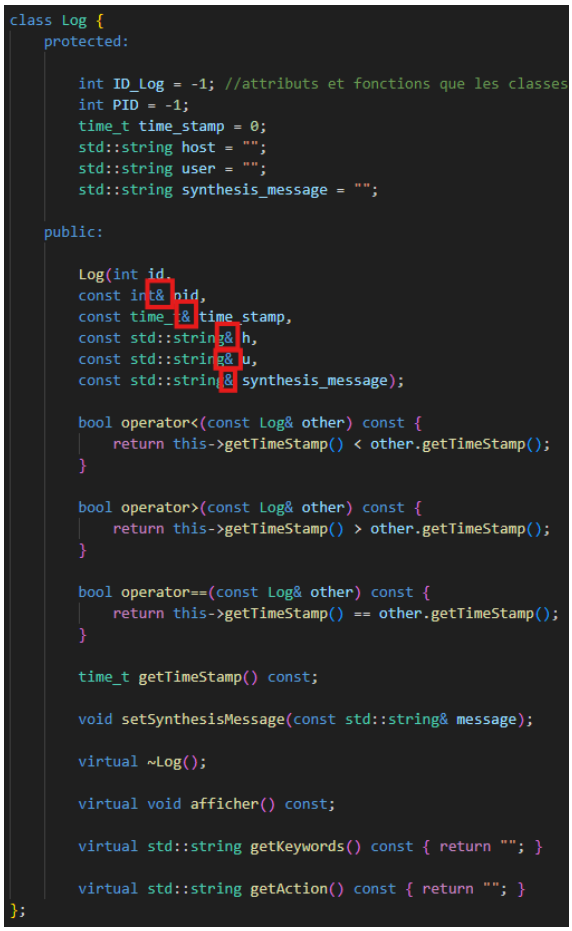
Pointeurs : Comme on peut (entre autres) le voir dans Log_Collector, on utilise par exemple des pointeurs pour créer des listes de logs de plusieurs types (Windows ou Linux) et pouvoir les gérer malgré cela.

Méthode const : Comme le montre ce *screen* issue de Log.cpp, on utilise la méthode const lorsque l'on cherche essentiellement a lire des données (par exemple pour les afficher), sans pour autant modifier la valeur des attributs qui sont dedans.



Transmissions par valeur :

Transmission par référence : dans ce screen venant de Log.hpp, on utilise le passage par référence pour éviter de travailler sur une copie des éléments.



```
class Log {
    protected:

        int ID_Log = -1; //attributs et fonctions que les classes
        int PID = -1;
        time_t time_stamp = 0;
        std::string host = "";
        std::string user = "";
        std::string synthesis_message = "";

    public:

        Log(int id,
        const int& pid,
        const time_t& time_stamp,
        const std::string& h,
        const std::string& u,
        const std::string& synthesis_message);

        bool operator<(const Log& other) const {
            return this->getTimeStamp() < other.getTimeStamp();
        }

        bool operator>(const Log& other) const {
            return this->getTimeStamp() > other.getTimeStamp();
        }

        bool operator==(const Log& other) const {
            return this->getTimeStamp() == other.getTimeStamp();
        }

        time_t getTimeStamp() const;

        void setSynthesisMessage(const std::string& message);

        virtual ~Log();

        virtual void afficher() const;

        virtual std::string getKeywords() const { return ""; }

        virtual std::string getAction() const { return ""; }
};
```