2️⃣

Cracking md5

Nelson JEANRENAUD - Théo MIRABILE

Conception

Classe PasswordCracker

Pour gérer la gestion du crackage de mot de passe nous avons décidé de créer une classe pour séparer la notion de crackage d'un mot de passe et la gestion de l'interface graphique (la barre de progression dans le cas de ce labo).

Champs

- La classe reçois en paramètre de son constructeur les informations de départ nécessaire pour la recherche (hash, salt, charset, nombre de caractères)
- Chaque instance de la classe va également stocker un référence sur chacun des PcoThreads utilisés pour la recherche. Ainsi que des informations sur l'état de la recherche. Comme le nombre de thread ayant terminé leur recherche, le nombre de possibilités déjà essayé, la progression de la recherche globale.

Recherche du mot de passe

Création des threads

La création des threads se déroule dans la méthode startCracking. Elle demande un paramètre entier non signé nbThreads sur lequel elle va boucler pour créer nbThreads PcoThread et les ajouter à la liste _threads.

Séparation du travail entre les threads

Pour séparer la recherche de manière optimisé entre les threads, nous avons décider de diviser le charset par le nombre de thread. À chaque thread sera assigner un

caractère du charset et son 1er mot de passe a essayer sera *nbChar* fois ce caractère. De cette manière, chaque thread s'occupe d'une partie des possibilités.

Par exemple, avec un charset donné [a,b,c,d,e,f,g,h,i] et un mot de passe à 4 caractères. Si le travail est assigné entre 3 thread la classe séparera le travail de la manière suivante :

1. [aaaa, dddd[
2. [dddd, gggg[
3. [gggg, iiii[

Le charset n'est pas toujours divisible par le nombre de thread. Pour gérer ces cas limite, le programme prend en note le reste de la division entière *(nombre de possibilités / nombre de thread)* et le distribue de manière équitable entre les threads.

Brut force

La recherche du mot de passe est la même que celle fournie dans la donnée du labo. Seul les paramètres son différents pour chaque thread.

Gestion de la progression

Comme déjà dit ci-dessus, séparer la notion d'interface utilisateur et de recherche de mot de passe était l'un des principaux objectifs de cette classe. C'est donc l'utilisateur de la classe qui a comme responsabilité de se charger la mise à jour de la barre de progression. Il peut le faire grâce au propriétés *getNbComputed, getNbToCompute* et *getProgress*.

Communication de la solution

Terminaison précoce des threads

Lorsque un thread a trouvé la valeur du mot de passe, il va annoncer la recherche comme terminée en modifiant le champ *_isResearchFinished.*

A chaque itération de la recherche, les threads vérifient que la recherche n'est pas encore terminée via ce champ. Il vont donc s'arrêter dès que le thread qui a trouvé la solution a modifié cette variable.

Résultat de la recherche

Le mot de passe trouvé est stocké dans le champ *_passwordFound*, accessible depuis l'extérieur de la classe avec un getter. Un utilisateur de la classe peut donc vérifier si la recherche est terminé et ensuite accéder a cette valeur. Le champ à la valeur NULL si la recherche n'est pas terminée ou que le mot de passe n'a pas pu être cracké.

Callback

Cette partie n'est pas demandé dans le cahier des charges du laboratoire. Mais nous avons trouvé intéressant d'utiliser cette notion dans la classe étant donnée qu'elle sera la méthode à utiliser dans un cas réel.

La classe fourni également la possibilité de définir une fonction de callback dans son constructeur. Cette fonction doit ne rien retourner et prendre en paramètre un pointeur sur un passwordCracker. Elle est appelé lorsque la recherche est marquée comme terminée et reçois en paramètre un pointeur sur l'instance du passwordCracker l'appelant.

Recherche infructueuse

Lorsque le hash fourni n'est pas crackable avec les informations fournis au threads, chaque thread retourne NULL. C'est alors le dernier thread a se terminer qui va marquer la recherche comme terminée et le résultat comme infructueux.

Cas limites et leurs solutions

Destructeur

Étant donnée que la recherche se fait dans une classe, les threads accède en lecture à des champs de l'instance. Il faut donc s'assurer que les threads soit bien terminées avant de détruire une instance de passwordCracker pour éviter que les thread accède a des emplacements mémoire non-alloués.

Pour se faire le destructeur de passwordCracker boucle sur tout les threads créés dans cette instance et demande leur terminaison via *PcoThread→requestStop()* chaque thread vérifie si son instance requiert qu'il se termine, et s'arrête instantanément si c'est le cas.

Le destructeur attend à l'aide de *PcoThread→join()* la terminaison de tout ses threads avant de se détruire.

Accès à des ressources partagées

Pour la gestion de l'avancement de la recherche, chaque threads incrémente la valeur de *_nbComputed* à chaque itération de la boucle de recherche. Il est donc très important de protéger cette section critique. Pour cela la classe comporte un mutex qui va permettre de sécuriser l'accès à cette variable.

Une deuxième section critique du programme est la gestion de la procédure de terminaison de la recherche. En particulier car la fonction de callback est appelé. Pour éviter que cette dernière soit appelé plusieurs fois il est également nécessaire de mettre en place un deuxième mutex pour garantir que cette section ne soit pas préemptée.