Université du Québec à Rimouski

Campus de Lévis

RAPPORT

TRAVAIL PRATIQUE #1

Par

Bastien Goulet et Cédrick Larrivée

Travail présenté à M. Martin Arsenault.

Dans le cadre du cours Sécurité informatique.

INF36207-MS

3 février 2023

Table des matières

[Algorithme 3](#_Toc1)

[Algorithme en détail 3](#_Toc2)

[Générer le jeton sur le client 5](#_Toc3)

[Validation du jeton sur le serveur 5](#_Toc4)

[Application client 6](#_Toc5)

[Application serveur 6](#_Toc6)

[Résultats 7](#_Toc7)

[Conclusion 7](#_Toc8)

[Annexes 8](#_Toc9)

[Références bibliographiques 9](#_Toc10)

# Algorithme

## Algorithme en détail

Pour éviter d’écrire du code redondant, nous avons décidé d’implémenter nos deux applications avec l’architecture en couche et en isolant le code de l'algorithme dans un projet à part. Cela nous a permis d’utiliser le même noyau pour nos deux applications. La génération des jetons est donc la même sur les deux applications. Pour créer le jeton, nous avons décidé de nous baser sur le temps écoulé depuis le premier janvier 1970. En d’autres termes, nous basons la génération du hash sur le Unix Timestamp, un nombre de secondes écoulées depuis le premier janvier 1970 théoriquement identique sur toute machine connectée au réseau.

En effet, il nous fallait un moyen constant et prédictible d’initialiser un *hash*. Le *Unix Timestamp* nous assure qu’il sera différent chaque seconde puisqu’il représente justement un nombre total de cette unité. En divisant cette valeur par la durée de vie du jeton en secondes, nous obtenons aisément une valeur entière unique pendant cette durée de temps. Cette valeur, appelée compteur par les différentes sources que nous avons consultées, devient l’une des entrées pour la fonction de chiffrement qui sert d’étape intermédiaire. Pourquoi utiliser une telle fonction de chiffrement alors que nous avons déjà un compteur en la personne du *Unix Timestamp* ? Pour introduire du pseudo-aléatoire dans le processus, bien entendu.

Les sources que nous avons consultées recommandaient une fonction de chiffrement utilisant l’algorithme SHA-1 qui produit un *hash* difficilement prédictible et nécessairement contenu sur 20 octets. Le compteur est la donnée à chiffrer et nous passons à cette fonction en plus un *salt*, qui est notre clé secrète. Dans le cadre d’une application en .NET, la classe *HMACSHA1* offre des méthodes pour chiffrer une donnée selon l’algorithme SHA-1. Grâce à ce procédé, nous avons à notre disposition, potentiellement chaque seconde (bien entendu, dans le cadre de ce travail, il s’agit plutôt de 60 secondes), un tableau de 20 octets différent dont les valeurs sont difficilement prédictibles. Il suffit désormais de choisir certains de ces octets et de les réorganiser afin de produire une valeur entière sur 8 caractères.

Nous ne prétendrons pas que l’algorithme ci-dessous est notre idée. En effet, elle est une adaptation de ceux présentés par nos sources en annexe.

La première étape consiste à choisir l’index de départ du premier octet parmi 4, dans le tableau, que nous utiliserons pour notre jeton. Nous choisissons 4 valeurs puisqu’une valeur entière sur 8 caractères, afin qu’elle puisse se rendre à sa valeur maximale, doit être contenue sur 32 bits, donc quatre octets. On prend le dernier octet du *hash* (hmacHash[^1]) et on applique le masque binaire *0000 1111* afin de récupérer la valeur des derniers 4 bits. Cette valeur est garantie d'être inférieure à 15 puisque 4 bits à 1 égalent 15. Ainsi, il est certain que l'on pourra manipuler 4 octets du tableau sans rencontrer une erreur *out of bound*.

On prend ensuite un des octets du tableau et on lui applique le masque binaire *0111 1111* afin d'être certain d'avoir un bit à gauche à zéro. Sur cette valeur nécessairement inférieure, on effectue un décalage à gauche de 24 bits (<< 24). L'octet est donc décalé vers le premier octet à gauche du 32 bits. Les 24 premiers sont remplis avec des zéros. Le masque 0x7F force le dernier bit à gauche à devenir un zéro. Ainsi, il est impossible d'obtenir une valeur de jeton négative puisque le dernier bit à gauche sera toujours zéro.

On répète ensuite cette même opération pour les 3 octets suivants dans le *hash* en modifiant le masque et le nombre de bits de décalage afin de positionner la valeur au bon endroit dans le 32 bits. Par exemple, pour le deuxième octet, on prend l'octet suivant du tableau et on lui applique le masque binaire *1111 1111* afin d'isoler notre octet dans le 32 bits. Sur cette valeur, on effectue un décalage à gauche de 16 bits (<< 16). L'octet est ainsi décalé vers le deuxième octet à partir de la gauche.

Le dernier octet n’est pas décalé, puisque l’application du masque 0xFF le maintient nécessairement dans le premier octet à partir de la droite. Il suffit ensuite d’effectuer un OU binaire entre les 4 valeurs afin de combiner les 4 octets sur un seul 32 bits et ainsi obtenir une valeur entière qui tient sur 32 bits.

Enfin, on calcule le reste de l'opération modulo entre notre valeur sur 32 bits et 10 à la puissance 8 (la longueur du jeton). Le reste sera nécessairement inférieur à 100 000 000, dans ce cas-ci, et nous aurons donc un jeton sur 8 caractères difficilement prédictibles.

## Générer le jeton sur le client

Pour générer un jeton pseudo-aléatoire, dans notre application client, nous devons commencer par initialiser les divers services requis avec l’aide d’interface. Les services counter, hash et otp nous seront utiles. Quand ce dernier sera initialisé, la méthode *ComputeNextOtp* sera appelée, le hash génèrera notre tableau d’octets qui sera passé à *ComputeOpt*. Ce dernier créera notre jeton aléatoire.

## Validation du jeton sur le serveur

Pour valider le jeton sur le serveur, ce dernier doit être généré de la même manière que sur notre application client puisque les deux applications utilisent le même noyau. À nouveau, durant l’initialisation de la fenêtre, la méthode *ComputeNextOtp* sera appelée. Le hash que nous avons passé au constructeur génèrera notre tableau d’octets qui sera passé à la fonction *ComputeOpt* qui comme décrit plus haut créera notre jeton aléatoire. Pour valider le jeton du client, ce dernier devra l’entrer dans le champ prévu à cet effet et ensuite appuyer sur le bouton Login. Cela va enclencher la validation du jeton qui va comme suit : le champ vérifie la longueur du jeton. S’il n’est pas égal à huit caractères, l’accès sera refusé. Nous allons tenter de convertir la chaine de caractère qui représente le jeton en un entier encore une fois, si cela ne fonctionne pas l’accès sera refusé. Finalement, une fonction du service otp sera appelée. Elle fera la validation du jeton de façon très simple. Pour ce faire, elle va accéder au jeton généré localement pour le comparer au jeton qui lui est passé en paramètre. La fonction retourne vraie si les deux jetons sont identiques. L’accès sera alors confirmé. Elle retournera la valeur faux s’ils ne le sont pas. Dans quel cas, l’accès sera refusé.

# Application client

L’application client est relativement simple d’utilisation. Il suffit de l’ouvrir et vous pourrez alors voir le temps restant avant la génération du prochain jeton. Toutes les 60 secondes, le compteur sera réinitialisé et un nouveau jeton sera généré. Sous le compteur se trouve le jeton actuel. Pour voir l’interface allez à l’annexe image 1.

# Application serveur

Tout comme l’application client, notre application serveur est aussi très simple d’utilisation. Un champ est présent pour entrer votre jeton qui doit être de 6 chiffres. Si le nombre de chiffres entré est plus long ou plus court, ou si des caractères autres que des chiffres sont présents alors l’accès sera refusé lors de l’utilisation du bouton Login qui permet de valider le jeton entré dans le champ. Finalement, le dernier jeton valide sera affiché sous le bouton de connexion.

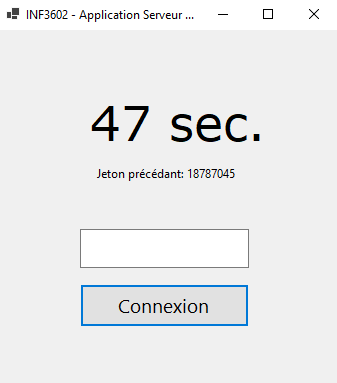
# Résultats

L’utilisation de l’architecture en couches était un bon choix. Il a grandement réduit le temps de développement, car il nous a permis de réutiliser notre code dans les deux applications tout en facilité l’intégration avec l’interface et ce dû à l’inversion des dépendances. Durant les tests, aucun bogue n’a été trouvé et les systèmes remplissaient bien leurs fonctions.

# Conclusion

En conclusion, nos applications, bien que simplistes, répondent bien aux divers critères qui étaient exigés. Notre application client génère un jeton pseudo-unique toutes les 60 secondes et l’afficher à l’écran avec le temps de vie reste. Notre serveur génère le même jeton pour pouvoir faire la validation. Les deux applications créent les mêmes jetons malgré le fait qu’ils ne communiquent pas ensemble. De plus, aucun des deux n’a besoin d’une connexion à internet pour fonctionner. En somme, elles sont totalement indépendantes l’une de l’autre.

# Annexes



# Références bibliographiques

[1] bizzehdeebizzehdee. “How to Get the Unix Timestamp in C#.” *Stack Overflow*, 1 July 1960, <https://stackoverflow.com/questions/17632584/how-to-get-the-unix-timestamp-in-c-sharp.>

[2] freeCodeCamp.org. “How Time-Based One-Time Passwords Work and Why You Should Use Them in Your App.” *FreeCodeCamp.org*, FreeCodeCamp.org, 5 June 2018, <https://www.freecodecamp.org/news/how-time-based-one-time-passwords-work-and-why-you-should-use-them-in-your-app-fdd2b9ed43c3/.>

[3] OgglasOgglas 56.9k3232 gold badges309309 silver badges387387 bronze badges, and NSKNSK 18,611 gold badge33 silver badges77 bronze badges. “Google Authenticator One-Time Password Algorithm in C#.” *Stack Overflow*, 1 Sept. 1964, <https://stackoverflow.com/questions/46442076/google-authenticator-one-time-password-algorithm-in-c-sharp.>

[4] Tdykstra. “C# Console App Template Changes in .NET 6+—.NET.” *C# Console App Template Changes in .NET 6+—.NET | Microsoft Learn*, <https://aka.ms/new-console-template.>