Sécurité

Anticheat

Pour limiter la triche, nous avons implémenter un anticheat. Chaque jeu possède ses propres règles, qui permettent de filtrer les acquisitions de scores frauduleuses.

Exemples :

- Dino: impossible de gagner plus de 2pt par secondes. Impossible de faire plus de 50 points.
- Moprion: impossible de faire plus d'une partie en 10s
- Juste prix: impossible de trouver le prix en moins de deux essais, plus d'une fois par minutes.

Comment ça fonctionne ?

Quand une partie est enregistrée, la date précise au moment de la sauvegarde du score est aussi enregitstrée. Cela permet à l'anticheat de récupérer les parties enregistrées dans la dernière minute pour appliquer les filtres:

```
now = datetime.datetime.now() - datetime.timedelta(minutes=1)
```

```
SELECT * FROM scores WHERE user="pseudo" date<=1736150504 ORDER BY date DESC
```

Cherche les parties de l'utilisateur, dont la date d'enregistrement est inférieure ou égale, à maintenant - 1 minutes: les parties enregisrées dans la dernière minute.

Les dates sont converis, en chiffres, suivant le timestamp: le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit.

Ensuite; le filtrage se fait au cas par cas, dans un switch case. `game` contient le jeu joué.

```
match game:
    case 'dino':
        # Logique de l'anticheat du dino
    case 'justeprix':
        # Logique de l'anticheat du justeprix
    # Ect
```

Exemple de filtre: le morpion

```
case 'morpion':
    # La partie doit durer 10s
    # <=> La différence des dates doit être supérieur à 10s
    # Une partie gagnée = 5 points, une partie égalité = 1, une partie perdue = 0
    points = points if points in [5, 1, 0] else 0
    return (now - last_game['date']).seconds > 10, points
```

Fonction complète dans le code :

```
def anticheat(game: str, points: int, user: dict):
    last_game = cursor.execute("SELECT * FROM scores WHERE user=? ORDER BY date
```

```
DESC", (user['pseudo'],)).fetchone()
    # Les parties de la dernière minute
    now = datetime.datetime.now()
    last_minute_games = cursor.execute("SELECT * FROM scores WHERE user=? AND
date<=?", (user['pseudo'], now - datetime.timedelta(minutes=1))).fetchall()
    last_minute_games = [build_score(row) for row in last_minute_games]
    if not last_game:
        return (True, points) if points < 25 else (False, 0)
    last_game = build_score(last_game)
    match game:
        case 'dino':
            # 1 point = 2 secondes
            # <=> Le nombre de secondes de la différences des dates / 2 doit être
supérieur au nombre de points
            return ((now - last_game['date']).seconds / 2) > points and points < 40,
points
        case 'morpion':
            # La partie doit durer 10s
            # <=> La différence des dates doit être supérieur à 10s
            # Une partie gagnée = 5 points, une partie égalité = 1, une partie
perdue = 0
            points = points if points in [5, 1, 0] else 0
            return (now - last_game['date']).seconds > 10, points
        case 'justeprix':
            # La partie doit durer 10s
            # <=> La différence des dates doit être supérieur à 10s
            # Ici points représente le nombre d'essais
            points = -0.5 * (points ** 2) + 30 if points <= 7 else 5  # Formule pour
les points: -0.5x²+30 et à partir de 7 essais, le score est constant à 5

            # Pas plus de 1 partie à 30 points dans une minute
            for game in last_minute_games:
                if game['game'] == 'justeprix' and game['points'] == 30:
                    return False, 0

            return (now - last_game['date']).seconds > 10, points
        case 'pfc':
            # La partie doit durer 5s
            # <=> La différence des dates doit être supérieur à 5s
            # Une partie gagnée = 5 points, une partie égalité = 1, une partie
perdue = 0
            points = points if points in [5, 1, 0] else 0
            return (now - last_game['date']).seconds > 5, points
        case 'osu':
            # TODO
            # 30s min
            # 15pt max
            ...
        case _:
            return False, 0
    return True, points
```

Sécurité BDD / mots de passes

Au début la BDD était publiée sur le dépôt de code (par soucis de collaboration). Cela pose des problèmes de sécurité (et de vie privée) puisque les données des utilisateurs sont donc publiques.

C'est pour cela que la base de "production" n'est pas publiée sur le dépôt.

Mots de passes

Comme décrit dans le cahier des charges, la sécurité des mots de passe est une couche de sécurité attendue. Nous utilisons l'algorithme PBKDF2: Password Based Key Derivation Function 2.

> *Une fonction de hachage cryptographique est une fonction qui, à une donnée arbitraire, associe une image fixe, pratiquement impossible à inverser. wikipedia.com*

En gros, pour chaque mot de passe possible il existe un hash; impossible à inverser : c'est comme si le mot de passe "en clair" était une pomme de terre en cuisine : une fois coupée en morceaux, impossible d'inverser le processus.

Son fonctionnement est itératif :

- il *hash* le mot de passe avec un algorithme donné un certain nombre de fois (SHA-256, 260 000 fois dans notre cas)
- en plus ce hashage, cet algorithme rajoute un sel ; une chaine de caractères aléatoires qui complique le cassage par force brute

Cet algorithme permet donc de hasher des mots de passes et de comparer leur hash mais rend presque impossible le cassage de ceux-ci, car il est lent...

L'implémentation python de cet algorithme est à `pbkdf2.py` et provient de Password hashing in Python with pbkdf2 - Simon Willison

Sécurité jetons

Notre système utilise les Json Web Token (JWT). Ces jetons uniques permettent d'identifier les utilisateurs.

Ils possèdent :

- une charge utile, avec le pseudo unique de l'utilisateur
- une date d'expiration
- une signature, qui permet de vérifier leur authenticité (si le jeton a bien été généré par notre site)

La signature ne peut être émise qu'a l'aide une clé secrète: n'importe quel personne ayant connaissance de la clé secrète peut générer des jetons qui font autorité (valides).

Cette clé, est générée aléatoirement^1 au lancement du site. Si elle venait à être compromise, le site doit donc "seulement" être relancé.

```
SECRET = secrets.token_urlsafe(512)
```

TODO: image JWT

[^1] à l'aide d'un RNG conçu pour la cryptographie docs.python.org

Protection des attaques XSS

> *Le cross-site scripting (abrégé XSS) est un type de faille de sécurité des sites web permettant d'injecter du contenu (javascript) dans une page, qui s'exécute sur les navigateurs web des utilisateurs visitant la page. [wikipedia.com]*

Ce type d'attaque consiste donc à entrer du code HTML (dont Javascript) dans un champ de texte affiché sur la page.

```
<p>
    {nom d'utilisateur}
</p>
```

Donc si le nom d'utilisateur est `<script>alert('Vulnérabilité')</script>` :

```
<p>
    <script>alert('Vulnérabilité')</script>
</p>
```

Et donc au chargement de la page, ce code sera exécuté sur le navigatur du visiteur.

Heureusement, nous utilisions Jinja, qui protège de ces attaques, en échappant les tags HTML: c'est à dire qu'elle affiche que tu texte.

`<script>alert('Vulnérabilité')</script>` ->
`<script>alert('Vuln&eacute;rabilit&eacute;')</script>`

Aussi, nous avons fait attention de ne pas utiliser des valeurs rentrées par l'utilisateur directement dans du Javascript (ou attributs `onclick` ect).

Injections SQL

SQL est un langage qui permet de faire des requêtes à une base de données.

```
SELECT * FROM users WHERE pseudo='unpseudo'
```

Récupère toutes les colonnes des lignes dont le pseudo est "unpseudo"

Cette requête peut être utile pour récupérer les informations enregistrées d'un utilisateur. Dans ce cas c'est avec une variable que l'on définira notre filtre `WHERE pseudo=`

```
pseudo = "unpseudo" # Exemple, on prend une valeur rentrée par l'utilisateur en
situation réelle
database.execute('SELECT * FROM users WHERE pseudo=\'' + pseudo + '\'')
```

Le problème est que cette valeur est une valeur renseignée par l'utilisateur, qui peut donc essayer d'échapper la chaine de caractère et faire sa propre requête à la base; c'est critique: il peut tout supprimer ou accéder aux informations des autres utilisateurs.

```
pseudo = "'; DROP TABLE users; --" # Exemple, d'exploitation de la faille
```

```
database.execute('SELECT * FROM users WHERE pseudo=\'' + pseudo + '\'')
```

```
SELECT * FROM users WHERE pseudo=''; DROP TABLE users; -- '
```

Supprime la base des utilisateurs !!!

La solution du problème : utiliser la fonctionnalité du module `sqlite` de Python, qui empêche toute possibilité d'injection. (et est maintenu régulièrement, au cas où une nouvelle faille serait découverte)

```
database.execute('SELECT * FROM users WHERE pseudo=?', (pseudo,))
```

Plus de problèmes; le module sqlite s'assurera que la chaine pseudo sera bien interprétée comme du texte.