**Namn:** Erling Blomberg

**Uppgift:** Salt pt1

**Flagga:** 210S{Click\_This}

**Svårighetsnivå:** 0,8

**Beskrivning:**

Klassen får fem filer: *db.json*, *main.py*, *salty.py*, *Your\_Task.md* och en gif. *Your\_Task* innehåller en kort ”story” som berättar vad uppgiften går ut på samt ger en liten ”nudge” i rätt riktning. Gif:en är lite kul men inte viktig för att lösa uppgiften.

De tre viktiga filerna som är själva uppgiften är *db.json, main.py* och *salty.py*. *db.json* skulle kunna ses som en godtycklig databas för användare. Den innehåller information för olika användare som namn, salt och ”information” som är krypterad. Det första man ser i main.py är att modulerna *json*, *base64* och *cryptography* importeras. Därefter följer funktionen *Salt* som importeras från *salty.py* som vi nu förstår är en modul.

Klassens uppgift är alltså att dekryptera informationen i *db.json* genom ”reverse-engineering” med utgångspunkt från *main.py.*

Fortsätter man att kolla i *main.py* ser man att det finns en nykel som leder till et skiffer baserat på Fernet funktionen från *cryptography* modulen. Därefter följer funktionen *encrypt* som tar två argument: *message* och *salt*. Funktionen *encrypt* har fyra steg:

1. En ny variabel *salted\_message* deklareras som summan an salt följt av *message*.

2. En ny variabel *encrypted\_message* deklareras som resultatet av kryptonen av *salted\_message* utifrån fernet skiffret.

3. en ny variabel *encoded\_ encrypted\_message* deklareras som strängen av bytesträngen *encrypted\_message.*

4. Returnerar *encoded\_ encrypted\_message*.

Sen kommer den klassiska *if \_\_name\_\_ == ”\_\_main\_\_”.* I *if* satsen ser man att *main.py* läser av *information.txt* en fil som inte klassen har tillgång till. *information.txt* innehåller en lista av strängar. En ny lista *encryption* deklareras och en *for* loop itererar över listan från *information.txt* som enumereras baserat på index får varje sträng (*index*) och strängen (*info*). I loopen deklareras en ny variabel *salt* som blir en sträng av *Salt* med argument 8. Vi tittar i *salty.py* och ser att funktionen *Salt* tar ett heltal som argument och itererar antalet gånger lika med argumentet, för varje iteration väljs 1 eller 0. Efter *while* loopen returneras en sträng konstruerad av varje valt värde. Detta innebär att variabeln salt får en åtta siffror lång kombination av 1 eller 0. Sedan deklareras en ny variabel *encrypted* som är lika med funktionen *encrypt* med argumenten *index* och *salt*. Efter det läggs en dictionary med key: ”user”+*index* och value: ytterligare en dictionary med key-valuepairs för *salt* och *info* till i listan *encryption*. Sist men inte minst skrivs *encryption* till filen *db.json.*

**Lösningskiss:**

För att lösa Salt pt1 behöver man göra alla steg i main.py i omvänd ordning. Först kopierar man nyckeln för *Fernet* skiffret för att konstruera samma skiffer. Sen läser man informationen från *db.json*. Därefter itererar man över alla användare och tar strängarna för *info* och *salt*. Omvandla stängen för *info* till en bytesträng med *base64* och använd den inbyggda funktionen *decrypt* från *cryptography* modulen för att dekryptera informationen. Sist men inte minst tar man bort antalet lika med saltets längd från första de karaktärerna i den resulterande strängen. Med ett skript som gör detta bör man få en länk som leder till en hemsida där flaggan visas.

**Namn:** Erling Blomberg

**Uppgift:** Salt pt2

**Flagga:** 210S{fisher}

**Svårighetsnivå:** 0,4

**Beskrivning:**

Salt pt2 är på många sätt väldigt lik Salt pt1. Det som skiljer är att det finns en ny *user* och kryptonen för info i *db.json* samt *main.py* och *Your\_Task.md*. I denna version av *main.py* importeras som tidigare *json* och funktionen *Salt* från *salty.py* men istället för *base64* och *cryptography* importeras *haslib* och *random* modulerna. Sedan följer en ny version an *encrypt* funktionen. Denna version har tre steg:

1. Samma som tidigare: en ny variabel *salted\_message* deklareras som summan an salt följt av *message*.
2. En ny variabel *hashed\_message* deklareras och blir strängen av *salted\_message* hashad med SHA-256.
3. Returnerar *hashed\_message*.

I *if* satsen deklareras en ny variabel *information* som är lika med innehållet från filen *information2.txt* som klassen åter igen inte har tillgång till. *Information2.txt* är identisk med *Information.txt* med undantaget att den nya flaggan har lagts till. Sedan blandas listan från *Information2.txt* så att klassen inte vet vart flaggan kommer vara när scripet avslutar och skriver till *db.json*. Resten av scriptet är identiskt med *main.py* från Salt pt1.

*Your\_task.md* förklarar den nya uppgiften: att hitta den nya flaggan genom att dekryptera *info* för *users* i *db.json* när informationen är hashad istället för krypterad med ett *Fernet* skiffer som lätt kan knäckas om man har nyckeln. Klassen får även veta att flaggan lagts till i *information2.txt* som annars är identisk med *Information.txt* samt att formatet på flaggan är 210s{xxxxxx} där varje x kan en godtycklig liten bokstav.

**Lösningskiss:**

För att lösa Salt pt2 tänkte jag mig att man utgår från Salt pt1. Eftersom vi vet att *Information2.txt* är identisk med *Information.txt* med undantag från den nya flaggan bör man använda sitt lösningsscript från Salt pt1 för att få listan av strängar i *Information.txt*. När man gjort detta bör man försöka matcha varje hashen för sträng från *Information.txt* för varje *users* *salt* i *db.json* mot hashen för en *users info*. Resultatet av detta är att man får en hash för en *user* där hashen för saltet och strängarna i *Information.txt* inte matchar, detta är flaggan.

Nästa steg är att skapa ett skript som genererar 6 karaktärer länga combinationer bland små bokstäver. Sätta in dessa i formen 210S{xxxxxx} och försöka matcha hashen för detta tillsammans med saltet från den udda *usern* mot den udda hashen. Detta lär ta mellan 50-60 sekunder att hitta flaggan om lösningen görs på linkande sätt som mitt förslag.

**Reflektion:**

Det som skiljer sig från min ursprungliga planering är att inga lösenord eller *info* för användare är lösta utan att man istället får scripten som krypterade databasen samt att jag använder hashing i Salt pt2. Detta påverkar även lösningsskissen då jag använder redan existerande metoder för kryptering istället för att skriva egna. Jag bestämde mig för att ändra uppgiften därför att den skulle varit brutalt svår och tidskrävande att lösa. Jag tycker att det är bättre att fler vågar testa och lyckas än att enbart de bästa ska få sig en utmaning.