PROGRAMMA

- Design pattern Authenticator
- Hashing di password (libreria Scrypt)
- Attacco DDoS e protezione (libreria Guava)
- Codice design pattern Authenticator

DESIGN PATTERN AUTHENTICATOR

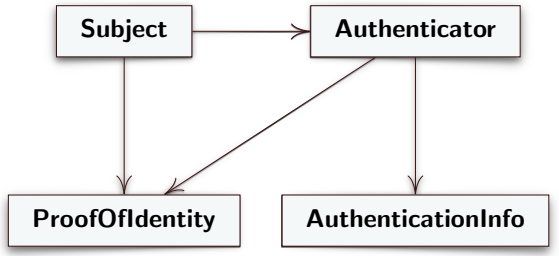
- Intento: permettere di verificare che il soggetto che intende accedere al sistema è chi dice di essere
- Contesto: i sistemi software contengono **risorse** che possono avere un valore in quanto includono: informazioni su affari, dati medicali, etc. Solo i soggetti che hanno qualche **motivo** per accedervi devono essere in grado di farlo
- Problema: come possiamo prevenire l'accesso da parte di impostori? La soluzione deve risolvere le seguenti forze
  - *Flessibilità*: una varietà di utenti richiede accesso al sistema e vi sono una varietà di parti, ciascuna con risorse a diversi livelli di **riservatezza**
  - *Dependability*: necessitiamo di autenticare gli utenti in modo **affidabile** e **sicuro**. Quindi usando un protocollo robusto in modo da proteggere i risultati dell'autenticazione

AUTHENTICATOR

- Problema (cont)
  - *Costo*: vi è un compromesso fra sicurezza e costo, più sicuro è il sistema e di solito più è costoso
  - *Prestazioni*: se l'autenticazione dovrà essere fatta frequentemente, le prestazioni sono da tener in conto
  - *Frequenza*: far sì che i soggetti **non** si debbano autenticare frequentemente
- Soluzione: usare un SINGOLO PUNTO DI ACCESSO che riceve le interazioni di un soggetto con il sistema ed applicare un protocollo per verificare l'identità del soggetto. Il protocollo usato può variare in base alle necessità dell'applicazione
- SINGOLO PUNTO DI ACCESSO definisce un solo posto da cui si può accedere al sistema, es. login dell'utente su sistema operativo

AUTHENTICATOR

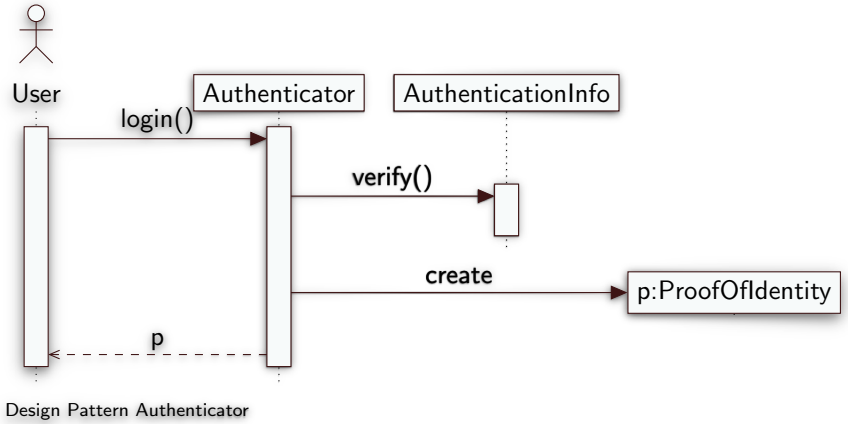
- Struttura della Soluzione
  - **Subject** (soggetto) richiede l'accesso al sistema
  - **Authenticator** riceve la richiesta e applica un protocollo che usa un'informazione nota, **AuthenticationInfo**, se l'autenticazione è soddisfatta crea una **ProofOfIdentity**, che è assegnata al soggetto per indicare che è legittimo



Design Pattern Authenticator

AUTHENTICATOR

- Soluzione per l'autenticazione dell'utente



Design Pattern Authenticator

AUTHENTICATOR

- Conseguenze
  - *Flessibilità*: al variare di protocollo (l'algoritmo di **Authenticator**) e informazione di autenticazione (**AuthenticationInfo**), si possono gestire tanti tipi di utenti autenticandoli in diversi modi
  - *Dependability*: poiché l'informazione sull'autenticazione è separata, possiamo **memorizzarla in un'area protetta**, alla quale tutti hanno accesso solo in lettura
  - *Costo*: possiamo usare una varietà di algoritmi e protocolli per autenticazione con diversa solidità. Le tre varianti esistenti includono: **ciò che l'utente conosce** (password), **ciò che l'utente possiede** (ID card), **ciò che l'utente è** (biometria)
  - *Performance e frequenza*: al momento della verifica dell'identità si può produrre una proof (**token**), così da non dover fare nuovamente l'autenticazione

AUTHENTICATOR

- Varianti
  - **Single Sign On** è un processo con il quale un soggetto verifica la propria identità ed il risultato della verifica può essere usato su vari domini per un certo intervallo di tempo
  - Il risultato dell'autenticazione è una Credenziale (ovvero identificativo, o **token**), usata per qualificare i futuri accessi dell'utente nel sistema distribuito

HASHING DI PASSWORD

- La password richiesta all'utente che si registra a un servizio deve essere successivamente usata per riconoscere l'utente (autenticazione)
- Non si può conservare la password in chiaro, ma si conserva un derivato della password (un codice hash) che è l'*AuthenticationInfo*
- La funzione di hashing dovrà essere
  - **deterministica**: lo stesso input dovrà dare lo stesso output
  - **irreversibile**: non consentire di trovare il dato a partire dall'hash
  - con output di lunghezza fissa, sebbene il dato in input sia di lunghezza variabile
  - far sì che un cambiamento piccolo del dato in ingresso dia hash completamente diversi

HASHING FUNCTION

- La **funzione di hashing** usata per le password dovrà essere **lenta** a eseguire in modo da rendere difficile (o impossibile) un **attacco *brute force***
  - Un attaccante che viene in possesso dell'hash potrebbe provare ad applicare la funzione di hashing tante volte (usando un dizionario) fino a trovare l'hash corrispondente
  - La lentezza della funzione rende difficile (molto lungo o impossibile) provare la funzione su un dizionario grande
- Un tempo di **100 ms** per la funzione di hashing è accettabile per un login e notevolmente **lungo per un attaccante**
  - Un dizionario di password può contenere 10^9 password, quindi per un attacco brute force occorre $t = 100 * 10^{-3} * 10^9 = 10^8$ s = 3 anni e 2 mesi circa

HASHING CON SALT

- L'aggiunta di *salt* (valore random) alla password fa sì che la funzione di hashing fornisca un hash sempre diverso persino quando due utenti hanno la stessa password
- Senza *salt* un attaccante potrebbe costruire un database compresso di hash di password, chiamato ***rainbow table***, che permette di recuperare velocemente le password
- Occorre conoscere il salt usato per l'hashing della password di un certo utente, in modo che durante la verifica delle credenziali venga usato lo stesso salt

LIBRERIA SCRYPT

- La funzione di hashing **Scrypt** (lettura: es crypt) è lenta, prende risorse di CPU e di memoria
- La funzione di hashing Scrypt ha tre parametri: N, r, p
  - *r* indica la lunghezza del risultato che sarà lungo *2 r k* bit, dove k-bit è la lunghezza dell'output dell'hashing function di base
  - *N* indica il lavoro da fare: la quantità di memoria e CPU usate sono dipendenti linearmente da N. Se N è una potenza di 2 l'operazione viene svolta senza sprechi. La memoria usata sarà *128 N r* bytes
  - *p* è un parametro di parallelizzazione; l'operazione di hashing può essere fatta sequenzialmente e occorrerà quindi p volte il tempo, oppure in parallelo, in tal caso occorrerà p volte la quantità di memoria a tempo costante

L'USO DI SCRYPT

- Per creare l'hash di una password `passwd`

```
final String hash = SCryptUtil.scrypt(passwd, 32768, 8, 1);
```

- Per verificare se una password `passwd` fa match con l'hash conservato (`authenticationInfo`)

```
SCryptUtil.check(passwd, authenticationInfo);
```

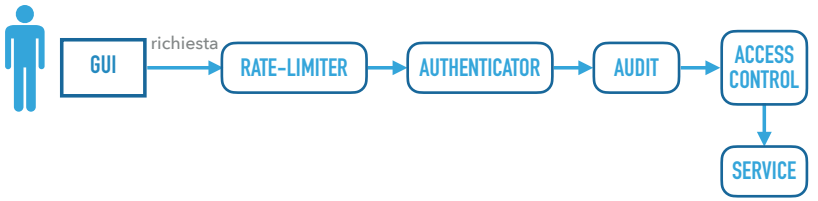
- All'interno del metodo `check`, il salt estratto dall'hash `authenticationInfo` verrà usato per l'hash function da applicare a `passwd`

ATTACCO DOS

- Un attacco Denial of Service (DoS) mira a ostacolare utenti legittimi a usare un servizio; spesso si attua generando un traffico enorme che sopraffa la capacità del server del servizio
- Un attacco distributed DoS (DDoS) usa tante macchine su internet per generare traffico, rendendolo più difficile da bloccare rispetto a un singolo client nocivo
- Molti attacchi DDoS si servono di qualche forma di amplificazione, come DNS amplification. Il client nocivo usa il DNS inserendo nella query al DNS un indirizzo di ritorno non vero (quello della macchina vittima). La macchina vittima sarà inondata di risposte a richieste DNS che non ha mai inviato

RATE-LIMITING

- La limitazione della frequenza di richieste fa sì che una richiesta non arrivi al servizio, se non si è sicuri che si abbiano risorse sufficienti a gestire la richiesta
- La verifica dell'identità può usare risorse, come pure gli altri strati di elaborazione, quindi il rate-limiting va fatto prima di ciascuna altra elaborazione



RATE-LIMITING

- Il componente che limita il rate di richieste dovrebbe essere applicato al livello del reverse proxy o gateway (punto di ingresso al lato server), quindi prima che le richieste arrivino ai server dei servizi
- La limitazione può essere aggiunta su ogni server, così da avere ulteriore sicurezza, in caso di gateway non ben funzionante o non ben configurato
  - **Defence in depth**: si hanno più strati di difesa così che il fallimento di un singolo strato non è sufficiente a compromettere l'intero sistema

GUAVA

- Guava fornisce l'implementazione di un componente rate-limiting tramite la classe `RateLimiter` che permette la definizione del rate di richieste al secondo da consentire (sotto sono 2 richieste al secondo)

```
RateLimiter rateLimiter = RateLimiter.create(2);
```

- Il richiedente che supera il limite verrà bloccato e messo in attesa

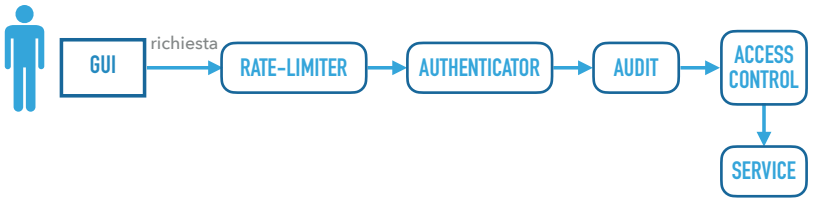
```
rateLimiter.acquire();
```

- Si può impostare un timeout di attesa ed espellere la richiesta

```
rateLimiter.tryAcquire(10, TimeUnit.MICROSECONDS);
```

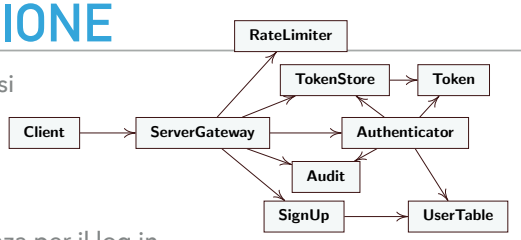
RIEPILOGO

- Lato server, per ciascuna richiesta, si dovrà
  - Limitare il rate di richieste
  - Verificare l'identità tramite credenziali o token
  - Verificare le autorizzazioni (vedi dopo)
  - "Processare" la richiesta

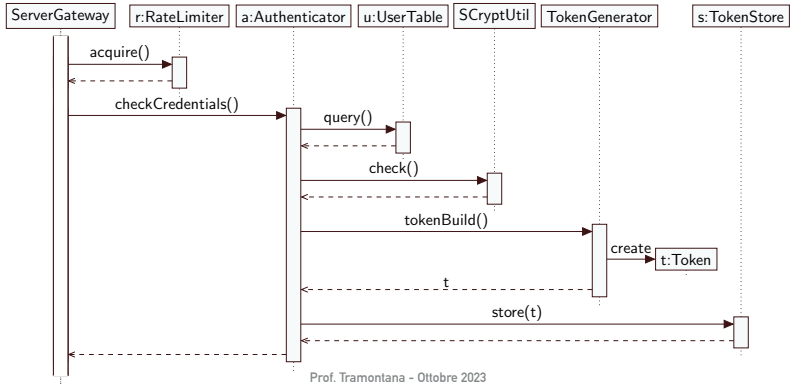


PROGETTAZIONE

- Diagramma delle classi

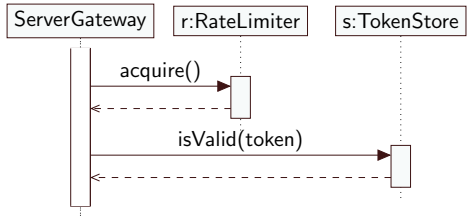


- Diagramma di sequenza per il log in



PROGETTAZIONE

- Diagramma di sequenza per richieste successive al log in



- vedere codice Authenticator