Spring Security

your gate to authentication and authorization

autumn 2016

Jan @Novoj Novotný

Organizer

Conference

Gold partner

General partner

Follow presentation with me at

http://novoj.github.io/reveal.js/spring-security.html

About me

- Java developer at FG Forrest
- YouTuber at Kafemlejnek.TV
- blogger at blog.novoj.net
- jOpenspace non-conference co-organizer
- co-author of MonkeyTracker service
- using Spring (Acegi) Security since 2007
- occassional speaker, father,
  MTB rider, runner ...

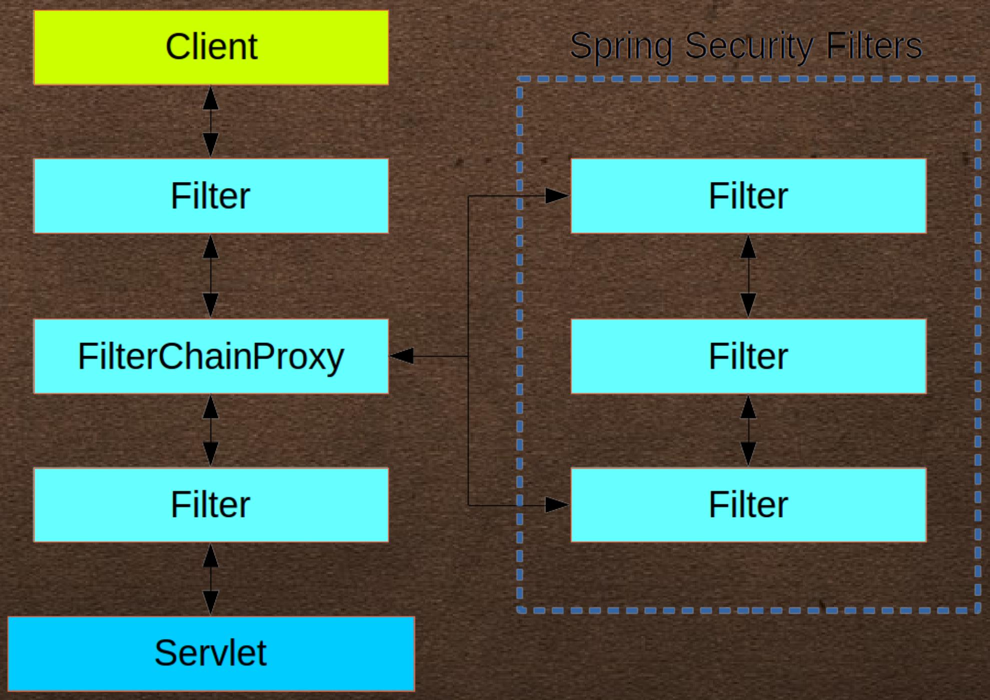
What Spring Security covers

1. **authentication**
    - Http Basic/Digest/x-509, LDAP, Kerberos, JAAS, OAuth, social networks
    - password generation / matching, remember me
2. **authorization**
    - url pattern matching
    - method call authorization, I/O data filtering
    - expression based / ACL based
3. **security negotiation / enforcement**
    - HTTP headers
    - HTTPS switching
    - Session management
    - CSRF protection
    - Cross-origin resource sharing
4. **test support**

Why should you consider it?

1. easy to implement
2. Spring integration (as you'd expect)
3. mature and widespread (Acegi started in 2003)
4. still in active development
5. easy extensible
6. well documented and tested

Architecture overview



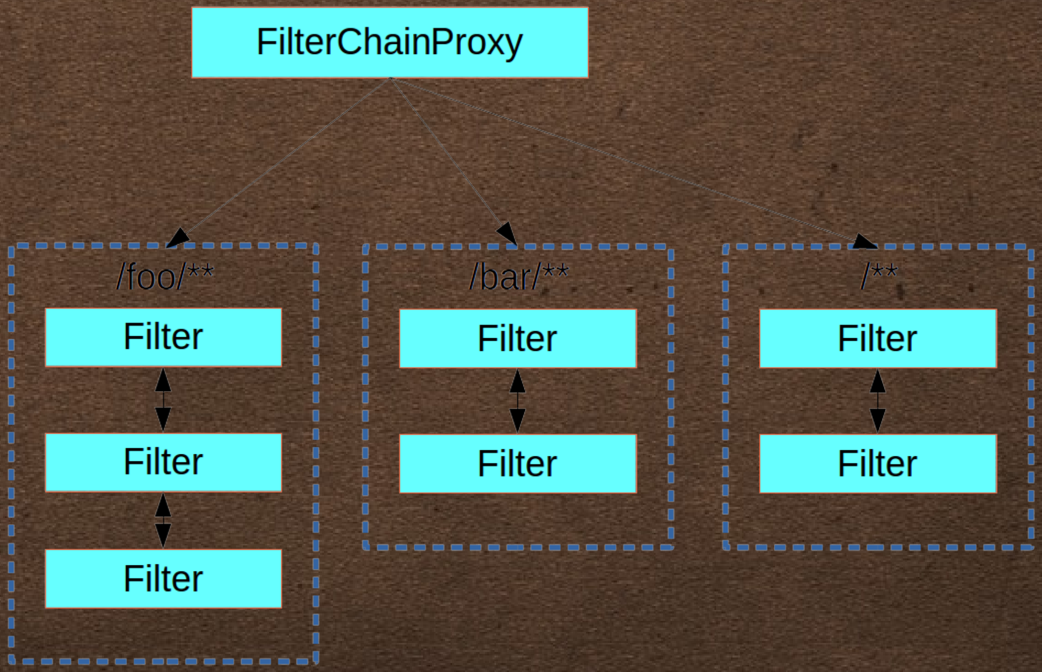
firewalls HTTP request / response
(directory traversal, http response splitting)

executes internal list of filters

Why?

- single definition in web.xml
  (even if it's complex inside)
- reliable "interruption"
  of every single request processing

Architecture overview



set of filters can differ for various path patterns

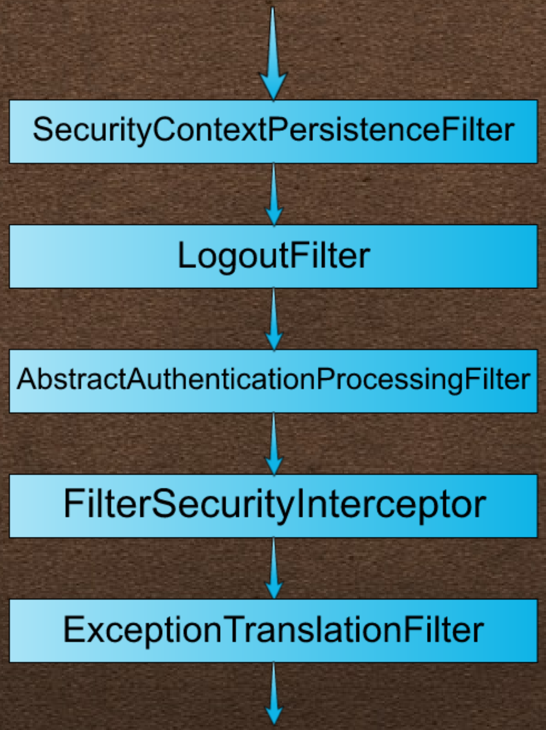
(org.springframework.security.web.SecurityFilterChain)

path matching is done via *RequestMatcher* mostly via *AntPathRequestMatcher* but also *RegexRequestMatcher* and more

```
/foo/*/bar/**/data/*.jpg
/foo/**
/bar/*
/*.html
```

Architecture overview

Standard filter composition


SecurityContextPersistenceFilter
LogoutFilter
AbstractAuthenticationProcessingFilter
FilterSecurityInterceptor
ExceptionTranslationFilter

1. loads SecurityContext from SecurityContext repository (usually HTTP session) and store it there also
2. monitors specific URL and logs a user out
3. does authentication (username, LDAP, OAuth ...)
4. handles AccessDeniedException and AuthenticationException navigating user to proper locations
5. examines each request against AccessDecisionManager and throws AccessDeniedException eventually
6. stores SecurityContext

Standard filters and their ordering

Architecture overview

Channel enforcement

ChannelProcessingFilter - relocates user from HTTP to HTTPS (and vice versa) when *ChannelDecisionManager* says so

HTTP headers management

HeaderWriterFilter - takes care of writing HTTP security headers to the response

- XXssProtectionHeaderWriter
- XFrameOptionsHeaderWriter
- HstsHeaderWriter
- StaticHeadersWriter (for instance CacheControlHeadersWriter)

Session management

ConcurrentSessionFilter - tracks presence of all sessions,
SessionManagementFilter - enforces session management

Configuration

Both types of configuration are similar one to another.

Security namespace

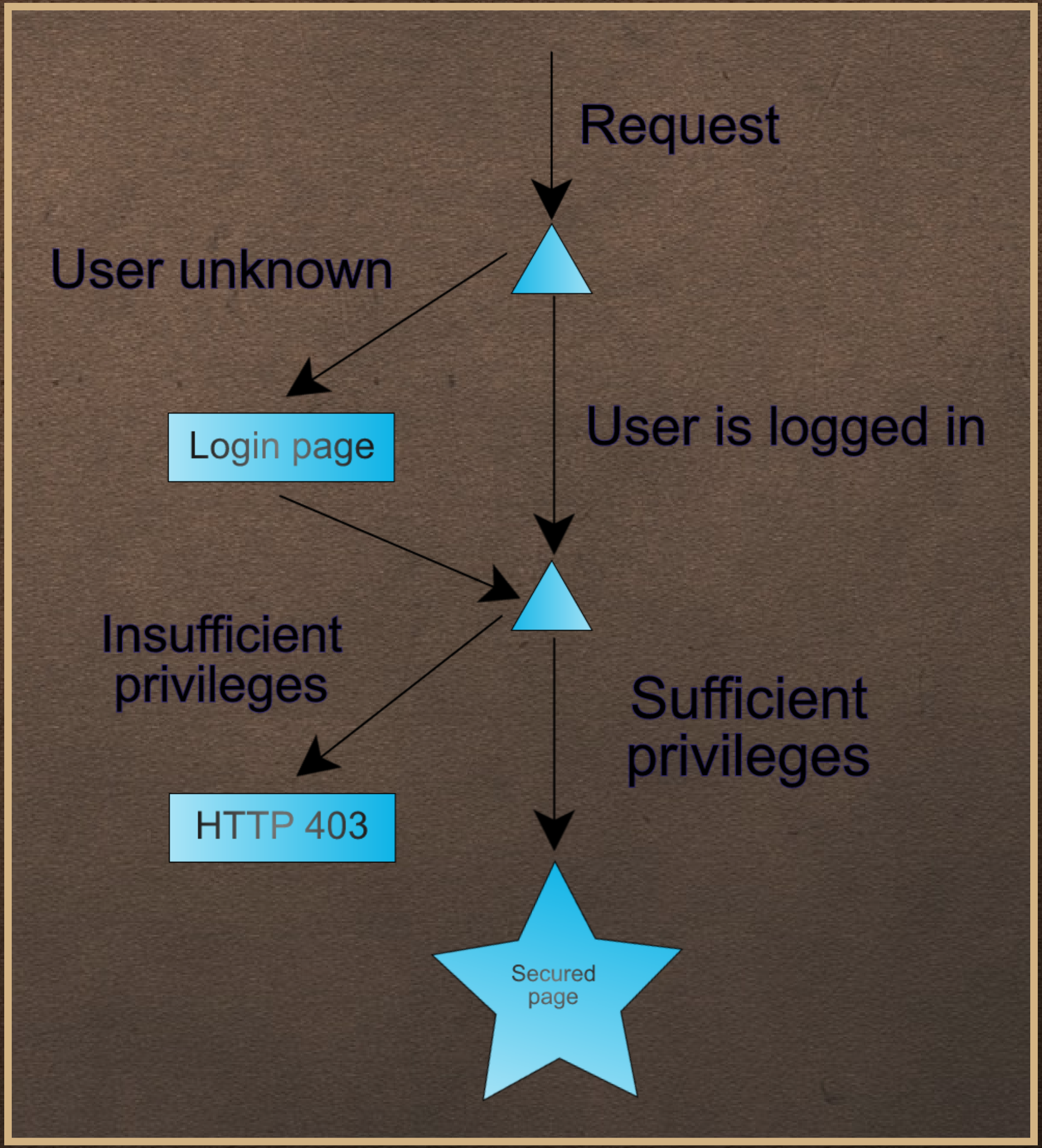
Click to show

Java configuration

Click to show

(Thanks to Remote plugin)

Authentication process



Login

- always use HTTPS protocol
- always use POST
- use CSRF token
- use the same error message when credentials don't match

Example

Login

What happens during login

Typically UsernamePasswordAuthenticationFilter uses AuthenticationManager that calls AuthenticationProvider (usually DaoAuthenticationProvider) to load user details from UserDetailsService implementation.

DaoAuthenticationProvider checks validity of the password via PasswordEncoder.

When password is ok, new Authentication object is created and stored in SecurityContext.

Subsequent requests

SecurityContextPersistenceFilter fills SecurityContextHolder (usually *ThreadLocal*) with SecurityContext from SecurityContextRepository

Logout

Recommendations

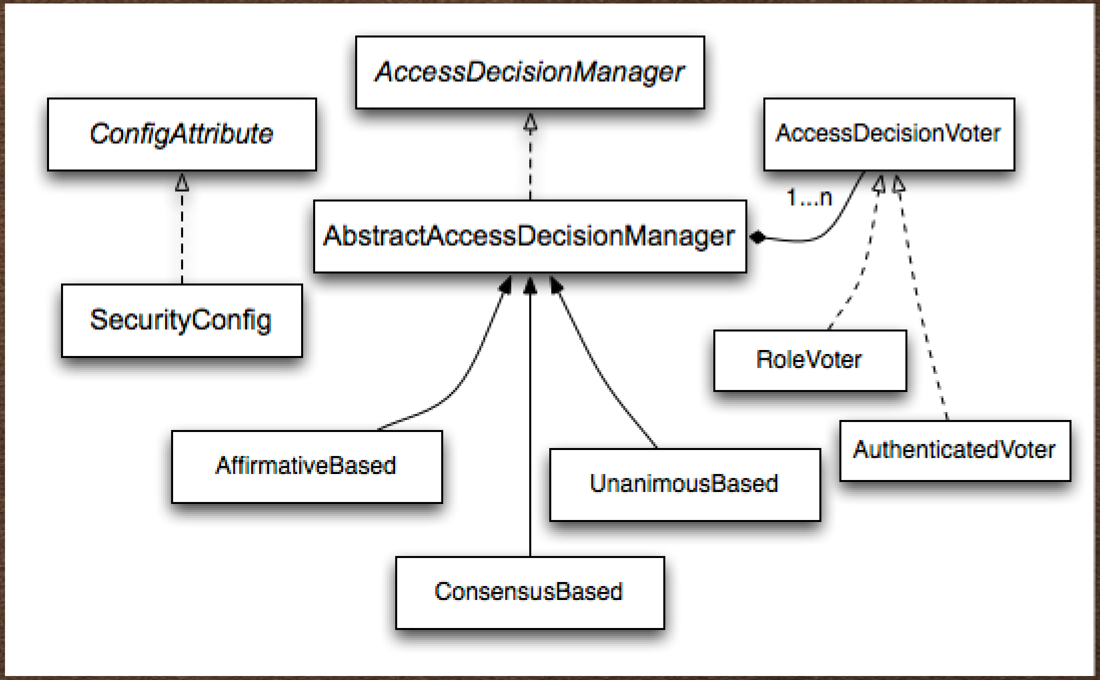
- always use POST
- use CSRF token

What happens on logout

- session is invalidated
- remember me tokens are cleared
- custom cookies might be cleared
- CSRF token is invalidated
- SecurityContext is emptied
- user is navigated to logout page

Example

Authorization



AccessDecisionManager

- AffirmativeBased:
  SINGLE GRANT -> GRANT
- UnanimousBased:
  SINGLE DENY -> DENY
- ConsensusBased:
  sum(GRANT) > sum(DENY) -> GRANT

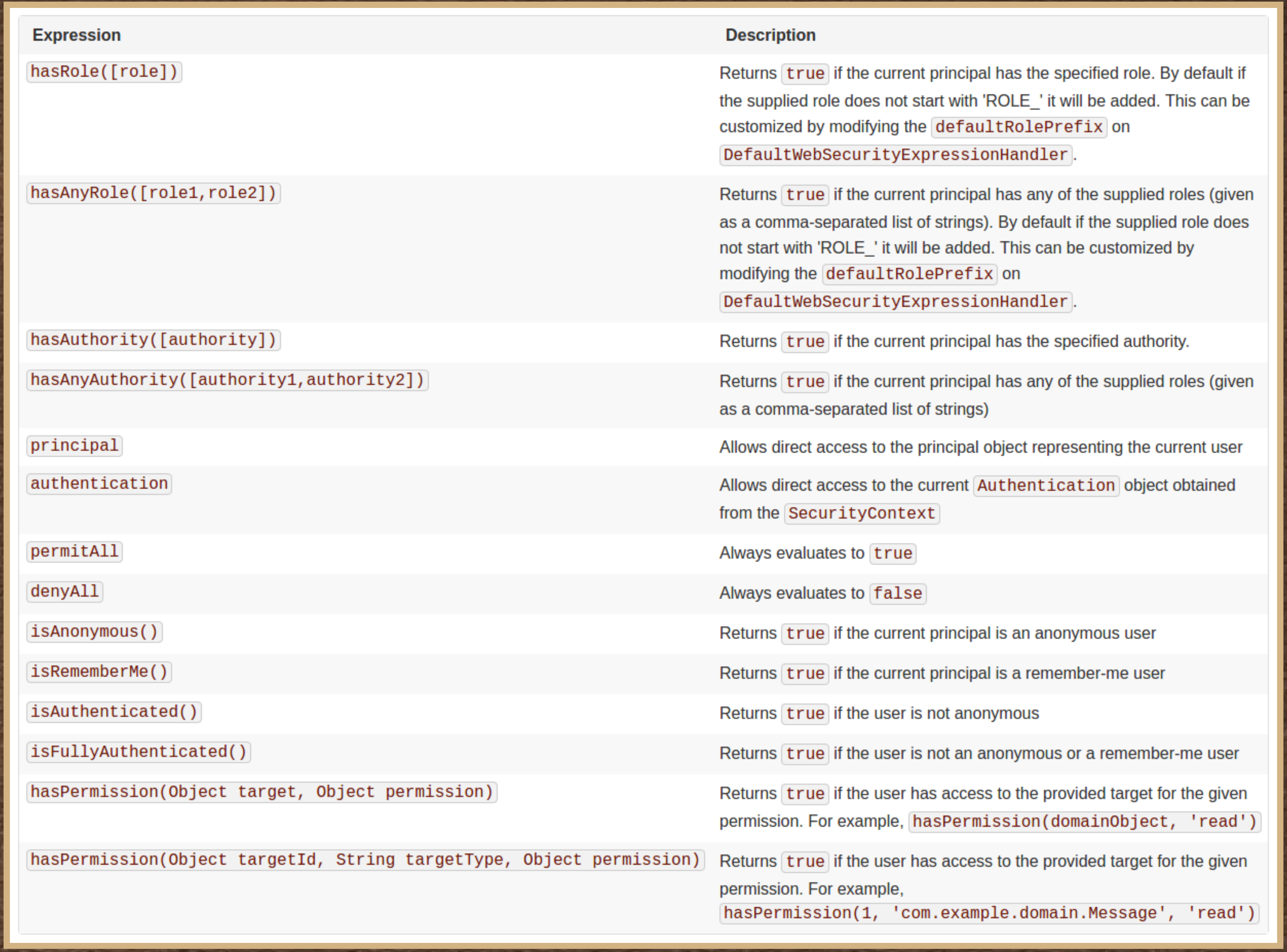
AccessDecisionVoter

- AuthenticatedVoter:
  IS_AUTHENTICATED_FULLY,
  IS_AUTHENTICATED_REMEMBERED,
  IS_AUTHENTICATED_ANONYMOUSLY
- RoleVoter (RoleHierarchyVoter):
  ROLE_ (GrantedAuthority)
- WebExpressionVoter:
  SpEL expression

SecurityConfig example:

```
/admin/**=ROLE_ADMINISTRATOR, ROLE_SUPER_ADMINISTRATOR
/userManagement/**=#{hasRole('ROLE_COMPANY_OWNER') and hasIpAddress('192.168.1.0/24')}
/**=IS_AUTHENTICATED_ANONYMOUSLY
```

Built-in-expressions



Expression	Description
`hasRole([role])`	Returns `true` if the current principal has the specified role. By default if the supplied role does not start with 'ROLE_' it will be added. This can be customized by modifying the `defaultRolePrefix` on `DefaultWebSecurityExpressionHandler`.
`hasAnyRole([role1,role2])`	Returns `true` if the current principal has any of the supplied roles (given as a comma-separated list of strings). By default if the supplied role does not start with 'ROLE_' it will be added. This can be customized by modifying the `defaultRolePrefix` on `DefaultWebSecurityExpressionHandler`.
`hasAuthority([authority])`	Returns `true` if the current principal has the specified authority.
`hasAnyAuthority([authority1,authority2])`	Returns `true` if the current principal has any of the supplied roles (given as a comma-separated list of strings)
`principal`	Allows direct access to the principal object representing the current user
`authentication`	Allows direct access to the current `Authentication` object obtained from the `SecurityContext`
`permitAll`	Always evaluates to `true`
`denyAll`	Always evaluates to `false`
`isAnonymous()`	Returns `true` if the current principal is an anonymous user
`isRememberMe()`	Returns `true` if the current principal is a remember-me user
`isAuthenticated()`	Returns `true` if the user is not anonymous
`isFullyAuthenticated()`	Returns `true` if the user is not an anonymous or a remember-me user
`hasPermission(Object target, Object permission)`	Returns `true` if the user has access to the provided target for the given permission. For example, `hasPermission(domainObject, 'read')`
`hasPermission(Object targetId, String targetType, Object permission)`	Returns `true` if the user has access to the provided target for the given permission. For example, `hasPermission(1, 'com.example.domain.Message', 'read')`

Web expression security

You can use either *simple rules* or *expression rules*, not both.

```
<http use-expressions="true">
    <intercept-url pattern="/user/{userId}/**"
                   access="@myAuthService.checkUserId(authentication,#userId)"/>
</http>
```

@ refers to Spring bean objects

# refers to url parameters

Method security

Operates using Spring AOP abstraction.
Thus you can configure it manually by custom Aspects or exact mapping.

Expression based security

Method annotations:

@PreAuthorize
@PreFilter
@PostAuthorize
@PostFilter

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

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

Filtering is not suitable for paged output.
But Spring Security can integrate with Spring Data!

Hiearachical roles

Useful in scenarios when user having certain role
should also have another role.

Example:

```
ROLE_ADMIN > ROLE_STAFF
ROLE_STAFF > ROLE_USER
ROLE_USER > ROLE_GUEST
```

Admin will be also STAFF, USER, GUEST regarding access decision logic.
Hierarchy rules can be changed without touching user data.

Permissions

If you need more robust authorization system - ie. when roles are not enough.

Implement your own PermissionEvaluator

```
public interface PermissionEvaluator extends AopInfrastructureBean {

    boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission);

    boolean hasPermission(Authentication authentication, Serializable targetId, String targetType, Object permission);

}
```

And configure it:

```
<security:global-method-security pre-post-annotations="enabled">
   <security:expression-handler ref="expressionHandler">
</security:expression-handler></security:global-method-security>

<bean id="expressionHandler" class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">
      <property name="permissionEvaluator" ref="myPermissionEvaluator"></property>
</bean>
```

Or use already implemented ACL system from Spring Security.

Meta annotations

Allows you to mitigate String based "programming".

Define meta-annotation

```
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
public @interface ContactPermission {}
```

Use meta-annotation

```
@ContactPermission
public void doSomething(Contact contact);
```

Currently doesn't allow multiple annotations on a single method.
Issue #4003

Security testing

Best practices ™

- **do test** the security in integration tests along with business logic
- prepare set of **personas** for each roles
- test **negative scenarios** that should finish with AccessDeniedException
- complex security expression extract to **access check method** or **PermissionEvaluator**, unit test these methods carefully

Method authorization

Test example

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@SecurityExecutionListeners
public class ExampleTestClass {

    @Test(expected = AccessDeniedException.class)
    @WithAnonymousUser
    public void getMessageUnauthenticated() {
         messageService.getMessage();
    }

    @Test
    @WithUserDetails("client@domain.cz")
    public void getMessageAsClientUser() {
        assertNotNull(messageService.getMessage());
    }

    @Test
    @WithUserDetails("admin@domain.cz")
    public void getMessageAsAdminUser() {
        assertNotNull(messageService.getMessage());
    }

}
```

Test meta-annotations

To avoid strings in annotations and add better maintainability use meta-annotations:

```
@Retention(RetentionPolicy.RUNTIME)
@WithUserDetails(value="admin@domain.cz")
public @interface RunAsAdministrator { }
```

And use it in test:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@SecurityExecutionListeners
public class ExampleTestClass {

    @Test
    @RunAsClient
    public void getMessageAsClientUser() {
        assertNotNull(messageService.getMessage());
    }

    @Test
    @RunAsAdmin
    public void getMessageAsAdminUser() {
        assertNotNull(messageService.getMessage());
    }

}
```

Spring MVC test support

```java
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
import static org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.*;
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class ExampleTestClass {

    @Autowired
    private WebApplicationContext context;
    private MockMvc mvc;

    @Before
    public void setup() {
        mvc = MockMvcBuilders.webAppContextSetup(context)
                             .apply(springSecurity())
                             .build();
    }

    @Test
    @WithAnonymousUser
    public void shouldUserLogin() {
        mvc.perform(formLogin("/auth").user("admin").password("pass")).with(csrf()))
           .andExpect(authenticated());
    }

    @Test
    @RunAsClient
    public void shouldUserLogout() {
        mvc.perform(logout("/signout")).with(csrf()))
           .andExpect(unauthenticated());
    }

}
```

User switching

Handy for developers as well as for customer administrators. Allows user with admin roles to switch to any other user with lower roles without knowing password.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    final SwitchUserFilter filter = new SwitchUserFilter();
    filter.setSwitchUserUrl("/switchUser");
    filter.setUsernameParameter("username");
    filter.setExitUserUrl("/restoreUserBack");
    http.addFilterAfter(filter, FilterSecurityInterceptor.class)
        .authorizeRequests()
            .antMatchers("/switchUser").hasAnyAuthority("ROLE_ADMIN")
            .antMatchers("/restoreUser").authenticated();
}
```

Consider also adding an IP check or other authentication checks to better secure user switching.

Warning! This feature is kind of a backdoor to your application.

Thank your for your attention

Contact me at @Novoj or novotnaci@gmail.com

Password management

Hygienic rules to follow:

- Use slow hashing function with unique salt per user (Spring Security default is Bcrypt)
- Don't send passwords in e-mails.
- Require strong passwords from users.
- Send login credentials only over SSL.
- When user changes his password / email require entering his current password.
- Do not load users with passwords (hashes) into memory.

Spring Security helps you with

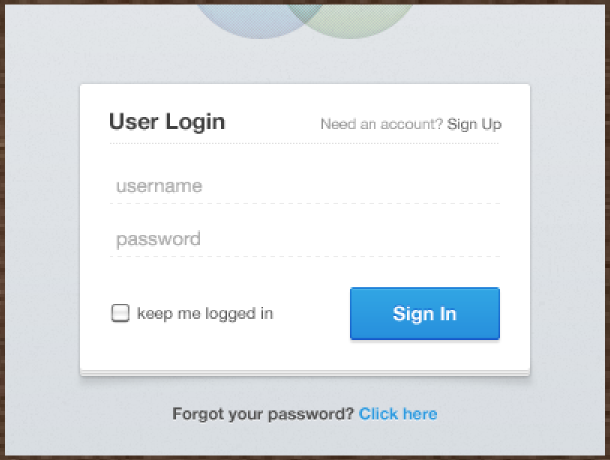
- hashing passwords
- enforcing SSL
- clearing pwd from logged in user object

Lost password facility

Hygienic rules to follow:

- If a user forgets their password, send them a secure one-time reset link, using a randomly generated reset token stored in the database. The token must be unique and secret, so hash the token in the database and compare it when the link is used.
- Do have same response when user exists as well as if he does not.
- Enforce that a token can only be used to reset the password of the user who requested it.
- Store token in database only in a "hashed" version.
- Once the token is used, it must be deleted from the database and must not be allowed to be used again.
- Have all password-equivilent tokens, including reset tokens, expire after a short time, e.g. 48 hours. This prevents an attacker exploiting unused tokens at a later date.
- Immediately display a form to allow the user to set a new password. Do not use temporary random generated passwords!
- Do not use "secret questions".

Remember me, hot or not?



Things to consider

- users do like it
- when stealed logs anyone in
- it opens gap for CSRF attack
- it may reveal original password

2-basic implementations

- Hash-Based Token (default)
- Persistent Token Approach

Remember me, hot or not?

Require interactive login for

- password change
- email change
- access to sensitive information (address)
- important actions (purchase)

Consider

- HTTPS over entire site
- secure only cookie (explicitly set necessary)
- http only cookie (explicitly set necessary)

```
/userProfile/*=IS_AUTHENTICATED_FULLY
/login=IS_AUTHENTICATED_ANONYMOUSLY
/logout=IS_AUTHENTICATED_ANONYMOUSLY
/lostPassword=IS_AUTHENTICATED_ANONYMOUSLY
/**=IS_AUTHENTICATED_REMEMBERED
```

Documentation

Hash-Based Token

```
base64(username + ":" + expirationTime + ":" +
md5Hex(username + ":" + expirationTime + ":" password + ":" + key))

username:          As identifiable to the UserDetailsService
password:          That matches the one in the retrieved UserDetails
expirationTime:    The date and time when the remember-me token expires, expressed in milliseconds
key:               A private key to prevent modification of the remember-me token
```

Disadvantages

- username + expiration time known to the attacker
- password is part of the hash
- MD5 cipher is very fast
- key is shared among all users

Persistent Token Approach

How it works

1. on successful login with Remember Me checked, a login cookie is issued
2. cookie consists of `series:token`
3. the series and token are unguessable random numbers from a suitably large space
4. data is stored together with user name in a database table
5. when a non-logged-in user visits the site with a login cookie, it is looked up in the database
6. if it is present, systems logs in a user with name stored with the pair in db
7. used token is removed from the database
8. new token is generated but series part is kept same, data are updated in DB and new cookie is sent to browser
9. if the username and series are present but the token does not match, a theft is assumed - user receives a warning and all of the user's remembered sessions are deleted
10. if the username and series are not present, the login cookie is ignored.

Disadvantages

- CSRF window is still open wide
- when stolen it opens up app to the attacker
- requires persistent storage
- if DB is leaked attackers can login as every user using RM cookie (see this article)

Advantages

- user name is not revealed
- password is not revealed
- when stolen, user gets informed on next login
- when stolen, attacker looses access on user next login

Cross-site request forgery

Technique that misuses an authenticated relation to execute operations via HTTP calls invisible to the user using well known URLs and parameters.

1. user logs in to an administration
2. user opens attacker site (in the same or different tab)
3. attacker site contains IMG/SCRIPT/STYLE elements that targets administration URL with appropriate parameters to execute actions

```
<img src="http://localhost:8080/gui/?action=add-url&s=http://evil.example.com/backdoor.torrent">
```

HTTP GET method is easiest to hack, other methods require social engeneering or JavaScript to execute. But are not hard to do.

CSRF protection

- Referer header check (not entirely recommended)
- requesting unique CSRF token in HTTP request (as parameter or header, not cookie!)

Originally Spring Security used CSRF token per request but it was quite impractical (browser back button, partial update) and thus CSRF token is now unique to session.

Things to consider

- CSRF protection needs to be part of login & logout process
- Javascript / AJAX calls might copy contents X-XSRF-TOKEN to HTTP header (attacker won't be able to do this due to *same origin policy*) ... see CookieCsrfTokenRepository
- Multipart form posts are harder to protect - entire body must be processed before CSRF protection might apply
- CSRF token SHOULD NOT be used in GET requests due to possible token leak via Referer header. CSRF token should be used only in HTTP POST actions that produces protected side effect.

Channel security

Allows you setup simple rules for SSL / plain HTTP protocol enforcement.

```
<http>
    <intercept-url pattern="/secure/**" requires-channel="https"></intercept-url>
    <intercept-url pattern="/css/**" requires-channel="http"></intercept-url>
    <intercept-url pattern="/**" requires-channel="any"></intercept-url>
    <port-mappings>
        <port-mapping http="9080" https="9443"></port-mapping>
    </port-mappings>
</http>
```

You don't have any excuse not to encrypt.

Security headers

These are defaults:

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

By only declaring:

```
<http>
        <headers defaults-disable="false"></headers>
</http>
```

X-Content-Type-Options

Header example:

```
X-Content-Type-Options: nosniff
```

Tells browsers (Chrome,IE) not to guess content type of the document from it's content.

Attack scenario: malicious user uploads PDF file that contains JavaScript. Your server let other users to download such file and open it in their browser. Even if server supplies PDF content type, browser might decide not to trust it and detect content type by the contents of the file. Finally it handles PDF file as JavaScript one and executes the script.

HTTP Strict Transport Security (HSTS)

Header example:

```
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

Tells browser that all resources on this domain must be reached by HTTPS protocol.

Except first request all subsequent ones will be loaded over HTTPS by the browser. Even if there is explicit link with HTTP protocol.

```
<img src="http://myserver.cz/img/someimage.jpg">
```

First request problem can be solved with HSTS Preload

HTTP Public Key Pinning (HPKP)

Header example:

```
Public-Key-Pins-Report-Only: max-age=5184000;
                             pin-sha256="d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=";
                             pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";
                             report-uri="http://example.net/pkp-report";
                             includeSubDomains
```

Tells browser that public key in the server certificate must match any of presented SHA256 hashes. Protects you against compromised CA.

Trust on First Use.

When using `Public-Key-Pins` and the server delivers an unknown public key, the client should present a warning to the user.

Reports can be processed with Report URI service

X-Frame-Options

Header example:

```
X-Frame-Options: deny
```

Tells browser not to load any FRAMES on your domain (or only frames targeting your/some domain).

Standardized as part of Content Security Policy.

Alternative variants:

```
X-Frame-Options: deny
X-Frame-Options: sameorigin
X-Frame-Options: allow-from www.seznam.cz
```

X-XSS-Protection

Header example:

```
X-XSS-Protection: 1; mode=block
```

Ask browser to assist you with reflected XSS attack.

If browser detects suspicious script in request that is reflected in the server response, script is blocked or reported.

Alternative variant:

```
X-XSS-Protection: 1;  report=https://report-uri.io/
```

Content Security Policy (CSP)

Header example:

```
Content-Security-Policy: script-src 'self' https://trustedscripts.example.com;
                         report-uri /csp-report-endpoint/
```

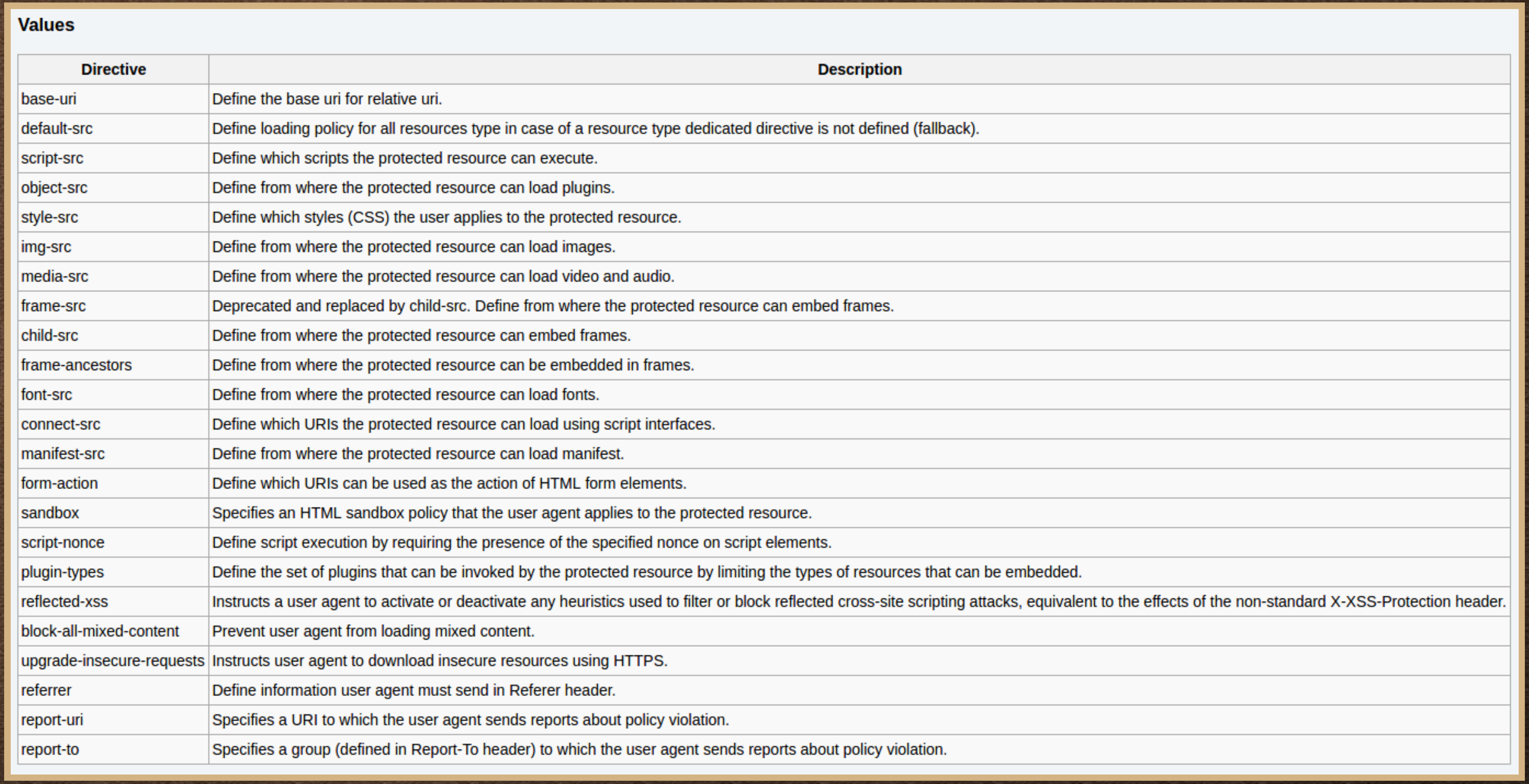
Site tels browser from which sources it expects to load resources.

Documentation at https://content-security-policy.com

Alternative variant:

```
Content-Security-Policy-Report-Only: script-src 'self';
                                     report-uri /csp-report-endpoint/
```

Content Security Policy (CSP)



Values

Directive	Description
base-uri	Define the base uri for relative uri.
default-src	Define loading policy for all resources type in case of a resource type dedicated directive is not defined (fallback).
script-src	Define which scripts the protected resource can execute.
object-src	Define from where the protected resource can load plugins.
style-src	Define which styles (CSS) the user applies to the protected resource.
img-src	Define from where the protected resource can load images.
media-src	Define from where the protected resource can load video and audio.
frame-src	Deprecated and replaced by child-src. Define from where the protected resource can embed frames.
child-src	Define from where the protected resource can embed frames.
frame-ancestors	Define from where the protected resource can be embedded in frames.
font-src	Define from where the protected resource can load fonts.
connect-src	Define which URIs the protected resource can load using script interfaces.
manifest-src	Define from where the protected resource can load manifest.
form-action	Define which URIs can be used as the action of HTML form elements.
sandbox	Specifies an HTML sandbox policy that the user agent applies to the protected resource.
script-nonce	Define script execution by requiring the presence of the specified nonce on script elements.
plugin-types	Define the set of plugins that can be invoked by the protected resource by limiting the types of resources that can be embedded.
reflected-xss	Instructs a user agent to activate or deactivate any heuristics used to filter or block reflected cross-site scripting attacks, equivalent to the effects of the non-standard X-XSS-Protection header.
block-all-mixed-content	Prevent user agent from loading mixed content.
upgrade-insecure-requests	Instructs user agent to download insecure resources using HTTPS.
referrer	Define information user agent must send in Referer header.
report-uri	Specifies a URI to which the user agent sends reports about policy violation.
report-to	Specifies a group (defined in Report-To header) to which the user agent sends reports about policy violation.

Session management

After successfull authentication all registered SessionAuthenticationStrategy are called. Preprogrammed implementations provide:

- session fixation protection
- concurrency control

Session fixation protection

Attack vector: Attacker will create it's own session in the application. Via social engineering sends link to a user (or uses similar technique) that will make user browser to use already existing attacker session. When user logs in, attacker can act on behalf of the user.

Defense

Don't accept session id from URL by configuring web.xml:

```
<session-config>
    <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

- COOKIE
- URL
- SSL (chapter 7.1.2)

Create new session on user login:

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

- newSession (clean)
- migrateSession (copy)
- changeSessionId (Servlet 3.1)

Concurrency control

Spring Security can track all sessions in SessionRegistry.

```
<session-management>
    <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" expired-url="/sessionExpired.html"/>
</session-management>
```

When *error-if-maximum-exceeded* is false and *max-sessions* is exceeded least recently used session is invalidated. User using this session will get *expired-url* page.

Thank your for your attention

Contact me at @Novoj or novotnaci@gmail.com