Building Layers of Defense with Spring Security

"We have to distrust each other.
It is our only defense against betrayal."

— Tennessee Williams

About Me

- Joris Kuipers (@jkuipers)
- Hands-on architect and fly-by-night Spring trainer @ Trifork

- `@author` tag in Spring Session's support for Spring Security

TRIFORK

Layers Of Defense

- Security concerns many levels
  - Physical, hardware, network, OS, middleware, applications, process / social, …
- This talk focuses on *applications*

TRIFORK

Layers Of Defense

- Web application has many layers to protect
- Sometimes orthogonal
- Often **additive**

Layers Of Defense

- Additivity implies some redundancy
- That's *by design*
- Don't rely on just a single layer of defense
  - Might have an error in security config / impl
  - Might be circumvented
  - AKA *Defense in depth*

Spring Security

- OSS framework for application-level authentication & authorization
- Supports common standards & protocols
- Works with any Java web application

Spring Security

Application-level:

- No reliance on container, self-contained
  - Portable
  - Easy to extend and adapt
- Assumes code itself is trusted

Spring Security

- Decouples authentication & authorization
- Hooks into application through interceptors
  - Servlet Filters at web layer
  - Aspects at lower layers
- Configured using Java-based fluent API

Spring Security Configuration

Steps to add Spring Security support:

1. Configure dependency and servlet filter chain
2. Centrally configure authentication
3. Centrally configure authorization
4. Configure code-specific authorization

Config: Authentication

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  /* set up authentication: */

  @Autowired
  void configureGlobal(AuthenticationManagerBuilder
                       authMgrBuilder) throws Exception

  {
     authMgrBuilder.userDetailsService(
        myCustomUserDetailsService());
  }

  // ...
```

Config: HTTP Authorization

```
/* ignore requests to these URLS: */

@Override
public void configure(WebSecurity web) throws Exception {
  web.ignoring().antMatchers(
              "/css/**", "/img/**", "/js/**", "/favicon.ico");
}


// ...
```

Config: HTTP Authorization

```
  /* configure URL-based authorization: */

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.POST,
                     "/projects/**").hasRole("PROJECT_MGR")
        .anyRequest().authenticated();
    // additional configuration not shown…
  }
}
```

Spring Security Defaults

This gives us:

- Various HTTP Response headers
- CSRF protection
- Default login page

HTTP Response Headers

“We are responsible for actions performed in response to circumstances for which we are not responsible”
— Allan Massie

TRIFORK

Disable Browser Cache

- Modern browsers also cache HTTPS responses
  - Attacker could see old page even after user logs out
  - In general not good for dynamic content
- For URLs not ignored, these headers are added

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

Disable Content Sniffing

- Content type guessed based on content
- Attacker might upload *polyglot* file
  - Valid as both e.g. PostScript *and* JavaScript
  - JavaScript executed on download
- Disabled using this header

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

Enable HSTS

- **H**TTP **S**trict **T**ransport **S**ecurity
- Enforce HTTPS for *all* requests to domain
  - Optionally incl. subdomains
  - Prevents man-in-the-middling initial request
- Enabled by default for HTTPS requests:

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

HSTS War Story

Note: *one* HTTPS request triggers HSTS for *entire domain and subdomains*

- Webapp might not support HTTPS-only
- Domain may host more than just your application
- Might be better handled by load balancer

Disable Framing

- Prevent *Clickjacking*
  - Attacker embeds app in frame as invisible overlay
  - Tricks users into clicking on something they shouldn't
- All framing disabled using this header
  - Can configure other options, e.g. SAME ORIGIN

```
X-Frame-Options: DENY
```

Block X-XSS Content

- Built-in browser support to recognize reflected XSS attacks
  - http://example.com/index.php?user=<script>alert(123)</script>
- Ensure support is enabled and blocks (not fixes) content

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

Other Headers Support

Other headers you can configure (disabled by default):

- HTTP Public Key Pinning (HPKP)-related
- Content Security Policy-related
- Referrer-Policy

CSRF / Session Riding Protection

"One thing I learned about riding is to look for trouble before it happens."

— Joe Davis

TRIFORK

Cross-Site Request Forgery

CSRF tricks logged in users to make requests

- Session cookie sent automatically
- Look legit to server, but user never intended them

Cross-Site Request Forgery

Add session-specific token to all forms

- Correct token means app initiated request
  - attacker cannot know token
- Not needed for GET with proper HTTP verb usage
  - GETs should be safe
  - Also prevents leaking token through URL

CSRF Protection in Spring Security

Default: **enabled for non-GET requests**

- Using session-scoped token
- Include token as form request parameter

```
<form action="/logout" method="post">
    <input type="submit" value="Log out" />
    <input type="hidden"
           name="${_csrf.parameterName}"
           value="${_csrf.token}"/>
</form>
```

CSRF Protection in Spring Security

- Doesn’t work for JSON-sending SPAs
- Store token in cookie and pass as header instead
  - No server-side session state, but still quite secure
  - Defaults work with AngularJS as-is

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.csrf()
        .csrfTokenRepository(
                CookieCsrfTokenRepository.withHttpOnlyFalse())
        .and()
    // additional configuration…
```

URL-based Authorization

“Does the walker choose the path, or the path the walker?”
— Garth Nix, Sabriel

URL-based Authorization

Very common, esp. with role-based authorization

- Map URL structure to authorities
  - Optionally including HTTP methods
- Good for coarse-grained rules

Spring Security Configuration

```
@Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      /* configure URL-based authorization: */
      .authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.POST,
                     "/projects/**").hasRole("PROJECT_MGR")

        // other matchers…
        .anyRequest().authenticated();
    // additional configuration not shown…
  }
}
```

URL-based Authorization

Might become bloated

- Esp. without role-related base URLs

```
http.authorizeRequests()
.antMatchers("/products", "/products/**").permitAll()
.antMatchers("/customer-portal-status").permitAll()
.antMatchers("/energycollectives", "/energycollectives/**").permitAll()
.antMatchers("/meterreading", "/meterreading/**").permitAll()
.antMatchers("/smartmeterreadingrequests", "/smartmeterreadingrequests/**").permitAll()
.antMatchers("/offer", "/offer/**").permitAll()
.antMatchers("/renewaloffer", "/renewaloffer/**").permitAll()
.antMatchers("/address").permitAll()
.antMatchers("/iban/**").permitAll()
.antMatchers("/contracts", "/contracts/**").permitAll()
.antMatchers("/zendesk/**").permitAll()
.antMatchers("/payment/**").permitAll()
.antMatchers("/phonenumber/**").permitAll()
.antMatchers("/debtcollectioncalendar/**").permitAll()
.antMatchers("/edsn/**").permitAll()
.antMatchers("/leads/**").permitAll()
.antMatchers("/dynamicanswer/**").permitAll()
.antMatchers("/masterdata", "/masterdata/**").permitAll()
.antMatchers("/invoices/**").permitAll()
.antMatchers("/registerverification", "/registerverification/**").permitAll()
.antMatchers("/smartmeterreadingreports", "/smartmeterreadingreports/**").permitAll()
.antMatchers("/users", "/users/**").permitAll()
.antMatchers("/batch/**").hasAuthority("BATCH_ADMIN")
.antMatchers("/label/**").permitAll()
.antMatchers("/bankstatementtransactions", "/bankstatementtransactions/**").permitAll()
.antMatchers("/directdebitsepamandate", "/directdebitsepamandate/**").permitAll()
.anyRequest().authenticated()
```

URL-based Authorization

Can be tricky to do properly

- Rules matched in order
- Matchers might not behave like you think they do
- Need to have a catch-all
  - `.anyRequest().authenticated();`
  - `.anyRequest().denyAll();`

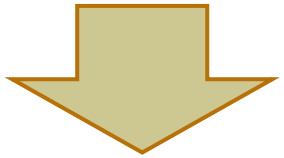
URL Matching Rules Gotchas

```
http.authorizeRequests()
  .antMatchers("/products/inventory/**").hasRole("ADMIN")
  .antMatchers("/products/**").hasAnyRole("USER", "ADMIN")
  .antMatchers(…
```

Ordering very significant here!

```
  .antMatchers("/products/delete").hasRole("ADMIN")
```

Does NOT match `/products/delete/` (trailing slash)!



```
  .mvcMatchers("/products/delete").hasRole("ADMIN")
```

TRIFORK

Method-level Authorization

"When you make your peace with authority,
you become authority"

— Jim Morrison

TRIFORK

Method-Level Security

- Declarative checks before or after method invocation
- Enable explicitly

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig
        extends WebSecurityConfigurerAdapter
{   …    }
```

@PreAuthorize Examples

```
@PreAuthorize("hasRole('PRODUCT_MGR')")
Product saveNew(ProductForm productForm) {
```

Refer to parameters,
e.g. for multitenancy

```
@PreAuthorize("hasRole('PRODUCT_MGR') &&
 #product.companyId == principal.company.id")
void updateProduct(Product product) {
```

@PostAuthorize Example

```
@PostAuthorize("returnObject.company.id ==
                principal.company.id")
Product findProduct(Long productId) {
```

Refer to returned object

Expressions in @Pre-/PostAuthorize

- Built-ins
  - hasRole(), hasAnyRole(), isAuthenticated(), isAnonymous(), …
- Can add your own…
  - Relatively complex

Expressions in @Pre-/PostAuthorize

- …or just call method on Spring Bean instead

```
@PreAuthorize("@authChecks.isTreatedByCurrentUser(#patient)")
public void addReport(Patient patient, Report report) {
```

```
@Service
public class AuthChecks {

  public boolean isTreatedByCurrentUser(Patient patient) {
        // ...
  }
```

Method-level Security

Support for standard Java **@RolesAllowed**

- Role-based checks only
- Enable explicitly

```
@EnableGlobalMethodSecurity(
   prePostEnabled = true, jsr250Enabled = true)
```

```
@RolesAllowed("ROLE_PRODUCT_MGR")
Product saveNew(ProductForm productForm) {
```

Programmatic Security

Easy programmatic access & checks

- Nice for e.g. custom interceptors
- Preferably *not* mixed with business logic

```
Authentication auth =
  SecurityContextHolder.getContext().getAuthentication();
if (auth != null  && auth.getPrincipal() instanceof MyUser) {
  MyUser user = (MyUser) auth.getPrincipal();
  // ...
```

Programmatic Use Cases

Look up current user to:

- Perform authorization in custom filter/aspect
- Populate Logger MDC
- Pass current tenant as Controller method parameter
- Auto-fill last-modified-by DB column
- Propagate security context to worker thread
- …

Access Control Lists

"Can't touch this"
— MC Hammer

TRIFORK

ACL Support

- Spring Security supports *Access Control Lists*
  - Fine-grained permissions per secured item
- Check before / after accessing item
  - Declaratively or programmatically
- Not needed for most applications

Defining ACLs

- Persisted in dedicated DB tables
- Entity defined by type and ID
- Access to entity per-user or per-authority
- Access *permissions* defined by int bitmask
  - read, write, delete, etc.
  - granting or denying

Checking ACLs

- Check performed against instance or type+id
- Multiple options for permission checks
- Using SpEL expressions is easy

```
@PreAuthorize("hasPermission(#contact, 'delete') or
               hasPermission(#contact, 'admin')")
void delete(Contact contact);


@PreAuthorize("hasPermission(#id, 'sample.Contact', 'read') or
               hasPermission(#id, 'sample.Contact', 'admin')")
Contact getById(Long id);
```

Other Concerns

“Concern should drive us into action, not into a depression.”
— Karen Horney

Enforcing HTTPS

- Can enforce HTTPS channel
  - Redirect when request uses plain HTTP
- HTTPS is usually important
  - Even if your data isn't
  - Attacker could insert malicious content
- Might be better handled by load balancer

Limiting Concurrent Sessions

- How often can single user log in at the same time?
- Limit to max nr of sessions
- Built-in support limited to single node
- Supports multi-node through Spring Session

Password Hashing

- Are you storing your own users and passwords?
- Ensure appropriate hashing algorithm
  - BCrypt, PBKDF2 & SCrypt support built in
  - Don’t copy old blogs showing MD5/SHA + Salt!

CORS

- Cross-Origin Resource Sharing
- Relaxes same-origin policy
  - Allow JS communication with other servers
- Server must allow origin, sent in request header
  - Preflight request used to check access:
    must be handled *before* Spring Security!

Enabling CORS Support

- Spring-MVC has CORS support
- For Spring Security, just configure filter

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .cors().and()
        // ... other config
```

- No Spring-MVC?
  Add `CorsConfigurationSource` bean

Conclusion

- Spring Security handles security at all application layers
- Combine to provide defense in depth
- Understand your security framework
- Become unhackable!
  - Or at least be able to blame someone else…