Spring Security

The Spring Boot Way

CYPRESS
DATA DEFENSE

Spring Boot & Spring Security

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Spring Boot
- Spring Security
- Spring Tool Suite
- Springline Demo

Spring Boot

Spring Boot allows applications to be built without the messy XML or Java configuration

Spring apps run as a .jar file

No .war files

Embedded Tomcat

http://projects.spring.io/spring-boot/

Spring 2 Example

In the beginning, there was Spring 2 XML config:

```
<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>
<http auto-config='true'>
   <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

Spring 3 Example

Then, there was Spring 3 Java config:

```
@Configuration
@ComponentScan(basePackageClasses = Application.class, includeFilters =
@Filter(Controller.class), useDefaultFilters = false)
class WebMvcConfig extends WebMvcConfigurationSupport {
@Bean
    public TemplateResolver templateResolver() {
        TemplateResolver templateResolver = new
           ServletContextTemplateResolver();
        templateResolver.setPrefix(VIEWS);
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");
        templateResolver.setCacheable(false);
        return templateResolver;
    }
}
```

Spring Boot Example

Introducing the best yet, Spring Boot:

```
@SpringBootApplication
public class SpringlineApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringlineApplication.class, args);
    }
}
```

Spring Boot Default Security

All paths are protected (/*)

No role-based or permission-based authorization

No login page

No backend user storage

Authentication over HTTP

Spring Boot Security Config

Ok, so we still have to configure Spring Security:

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // http configuration settings go here…
    }
}
```

Spring Security 4

Spring Security 4.1.1

Built-in support for:

- Authentication, authorization, validation, crypto
- CSRF, XSS, clickjacking, password management

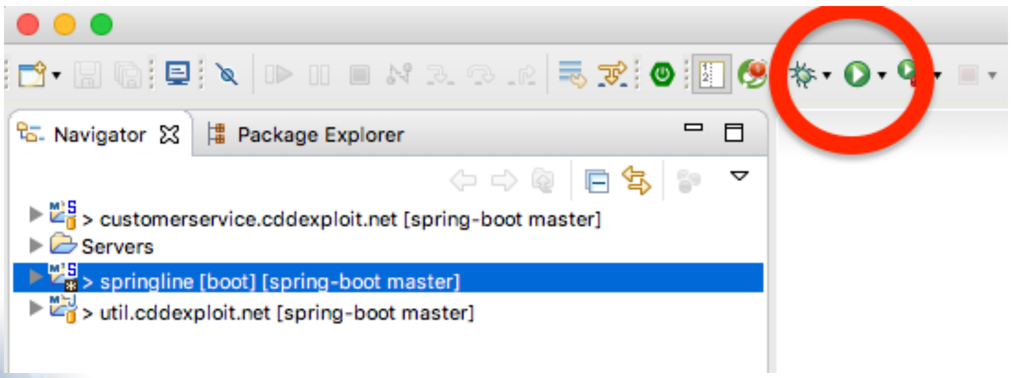
http://docs.spring.io/spring-security/site/docs/4.1.1.RELEASE/reference/html/

Spring Tool Suite

The Spring Tool Suite (STS) provides an Eclipse-based IDE for creating and developing spring based apps:

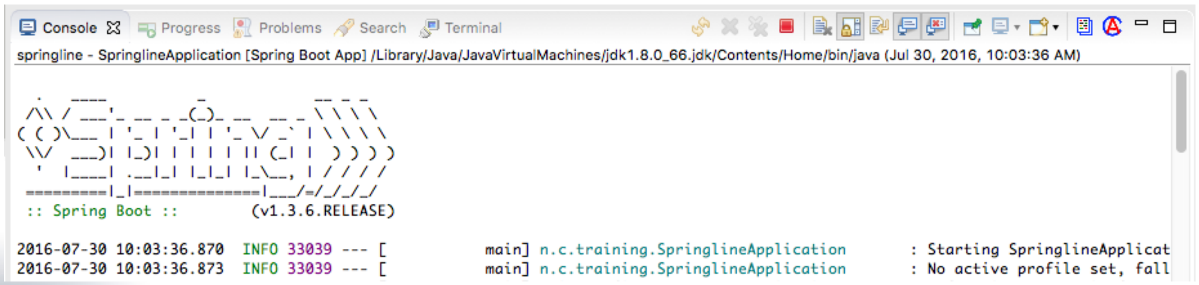
https://spring.io/tools

Spring Tool Suite | Running Springline



> customerservice.cddexploit.net [spring-boot master]
Servers
> springline [boot] [spring-boot master]
> util.cddexploit.net [spring-boot master]

Spring Tool Suite | Console



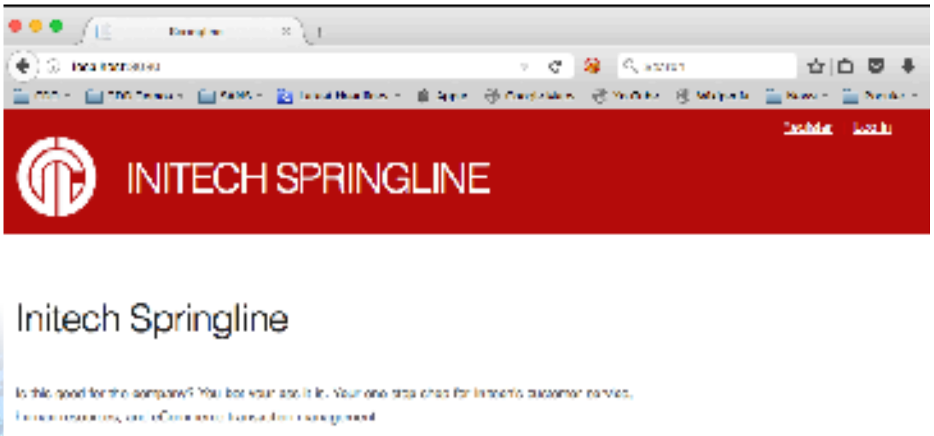
```
springline - SpringlineApplication [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java (Jul 30, 2016, 10:03:36 AM)

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.3.6.RELEASE)

2016-07-30 10:03:36.870  INFO 33039 --- [           main] n.c.training.SpringlineApplication       : Starting SpringlineApplicat
2016-07-30 10:03:36.873  INFO 33039 --- [           main] n.c.training.SpringlineApplication       : No active profile set, fall
```

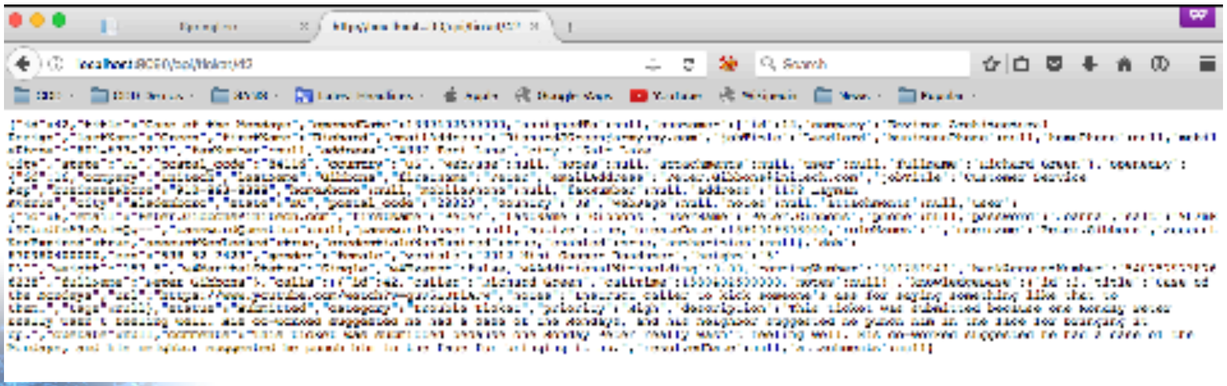
Spring Tool Suite | Springline Web

http://localhost:8080



Spring Tool Suite | Springline Services

http://localhost:8090/api/ticket/42



```
[illegible]
```

Exercise 0: Springline

Open the Spring Tool Suite (STS) app

Select the Springline_Workspace

Import the 3 maven projects

Find the @SpringBootApplication classes

Examine the Spring Security config files

Launch the springline services and web apps

Spring Security | Authentication

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Authentication Modes
- Form Login Configuration
- Password Encoder
- Multi-Factor Authentication

Spring Authentication Manager

SecurityConfig.java

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
   //Configure auth here
}
```

Spring Authentication Modes

Basic

In Memory

LDAP

Custom Authentication Service

Basic Auth Configuration

Basic authentication is still commonly used to authentication systems and REST endpoints:

- Base64 encodes the username and password in the authorization header
- Requires apps to enforce HTTPS for all requests

To enable, add the BasicAuthenticationFilter to your filter chain

In Memory Configuration

User and role info is stored in memory

Does not scale for production apps

SecurityConfig.java:

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

LDAP Configuration

Leverages backend LDAP instance

http://docs.spring.io/spring-security/site/docs/4.1.1.RELEASE/reference/htmlsingle/#ldap-authentication-2

SecurityConfig.java:

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
   auth
      .ldapAuthentication()
      .userDnPatterns("uid={0},ou=users")
      .groupSearchBase("ou=groups");
}
```

Custom Auth Service Configuration (1)

Custom authentication against backend database

SecurityConfig.java:

```
@Inject
private UserDetailsService userService;

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
   auth
      .eraseCredentials(true)
      .userDetailsService(userService);
}
```

Custom Auth Service Configuration (2)

Customer Service > UserService.java:

```
@Service
@Transactional
public class UserService implements UserDetailsService, Serializable {
   @Override
   public UserDetails loadUserByUsername(final String username) {
      final User user = userRepository.findByUserName(username);
      […]
      List<GrantedAuthority> auth =
         AuthorityUtils.commaSeparatedStringToAuthorityList(user.getRoleNames());

      return new org.springframework.security.core.userdetails.User(
         username,  user.getPassword(), auth);
   }
}
```

Form Configuration

SecurityConfig.java:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .formLogin()
      .loginPage("/account/login")
      .permitAll()
      .failureUrl("/account/login?error=1")
      .loginProcessingUrl("/account/authenticate")
      .defaultSuccessUrl("/home")
}
```

Password Encoders

Spring Security ships with many different password encoding options:

- NoOpPasswordEncoder
- StandardPasswordEncoder
- Pbkdf2PasswordEncoder
- SCryptPasswordEncoder
- BCryptPasswordEncoder

NoOpPasswordEncoder

NoOpPasswordEncoder

- Password encoder that does NOTHING.
- Passwords stored in cleartext
- Only use in testing scenarios when cleartext passwords are required

StandardPasswordEncoder

StandardPasswordEncoder

- Uses SHA-256
- Random 8-byte salt
- 1024 iterations (i.e. work factor)
- System-wide secret used to provide additional protection
- Default password encoding method

Pbkdf2PasswordEncoder

Pbkdf2PasswordEncoder

- Uses PBKDF2
- Random 8-byte salt
- Configurable number of iterations (360,000 default)
- Configurable hash bytes (160 default)

SCryptPasswordEncoder

SCryptPasswordEncoder

- Uses SCrypt Bouncy castle implementation
- Configurable CPU cost parameter (16,348 default)
- Configurable memory cost parameter (8 default)
- Performs poorly in Java
- Not recommended by Spring Security for usage

BCryptPasswordEncoder

BCryptPasswordEncoder

- Configurable strength parameter between 4 and 31 (8 default)
- Configurable SecureRandom instance
- Recommended by Spring Security

Password Encoder Configuration

Configuring the password encoder type

SecurityConfig.java:

```
@Bean
public PasswordEncoder passwordEncoder() {
   return NoOpPasswordEncoder.getInstance();}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
   auth
      .eraseCredentials(true)
      .userDetailsService(userService)
      .passwordEncoder(passwordEncoder());}
```

Multi-Factor Authentication

- Passwords are a weak way to authenticate a user
  - E.g. RockU, Adobe, LinkedIn, Ashley Madison, Forbes, Snapchat, etc.
- Multi-factor (aka 2-factor) authentication requires users to enter an additional value

Spring Multi Factor Authentication

Don't bother looking, it's not built into Spring Security

Spring Security TOTP

Written by Ralph Schaer

Time-based One-Time Password (TOTP)

https://github.com/ralscha

Exercise 1: Springline Password Encoding (1)

The Springline application is storing passwords in cleartext

Your goal is to change Springline to hash passwords with the BCryptPasswordEncoder

Exercise 1: Springline Password Encoding (2)

1) Sign in to Springline

   Username: mwaddams, Password: Stapler

2) Locate the password encoder configuration in the SecurityConfig.java file

3) Open MySQLWorkbench and confirm the passwords are stored in the customer_service user table in cleartext

Exercise 1: Springline Password Encoding (3)

4) Use the encodePassword test stub in PasswordEncoderTests.java to hash the password for “mwaddams” and “blumbergh”
5) Update the customer_service user table and change the password values to the BCrypt hash
6) Change the Springline configuration to use the BCryptPasswordEncoder and verify you can login

Spring Security | Cryptography

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Key Generator
- Encryptor

Crypto Modules

Spring Security ships with a few different crypto modules:

Key Generators

Encryptors

Key Generators

Key generator classes provide a wrapper around SecureRandom

Simple way to generate keys, IVs, tokens, secrets, etc.

Default random value length is 8 bytes

Implementation found in the *BytesKeyGenerator* class

BytesKeyGenerator Examples

Default key generator producing a random 8 byte secret:

```
BytesKeyGenerator generator = KeyGenerators.secureRandom();
byte[] rnd = generator.generateKey();
```

Custom key generator producing a random 16 byte hex string:

```
BytesKeyGenerator generator = KeyGenerators.secureRandom(16);
String strRnd = new String(Hex.encode(generator.generateKey()));
```

Encryptors

Encryptor classes provide a symmetric encryption wrapper

Default algorithm is AES-256

Key is derived from password and salt values using PBKDF2

Random 16 byte IV values are applied as each message is encrypted

TextEncryptor Example

Example using the *BytesKeyGenerator* and *TextEncryptor* classes to encrypt a value:

```
String value = "my secret value";

//DON'T HARD-CODE IN REAL LIFE - USE A SECRETS MANAGER!!!
String secret = "my secret key";

//Generate salt for key derivation operation
BytesKeyGenerator random = KeyGenerators.secureRandom(16);
String salt = new String(Hex.encode(random.generateKey()));

TextEncryptor encryptor = Encryptors.text(secret, salt);
String cipherText = encryptor.encrypt(value);
```

Exercise 2: Spring Services Encryption (1)

The Springline Services application is storing SSNs in cleartext

Your goal is to change the Employee database entity to encrypt and decrypt SSNs using the *TextEncryptor*

Exercise 2: Spring Services Encryption (1)

1) Sign in to Springline

   Username: mwaddams, Password: Stapler

2) View the My Profile page and observe the SSN in the browser
3) Open MySQLWorkbench and confirm Milton's SSN is stored in the customer_service employee table in cleartext.

   Search for last_name = 'Waddams'

Exercise 2: Spring Services Encryption (2)

4) Use the encryptTextTest() test stub in EncryptorTests.java to encrypt the SSN employee id for "Waddams", "Lumbergh", and "Gibbons"

   Set the "secret" password for PBKDF

   Enter a SSN value

   Execute the encryptTextTest() stub for each user

5) Update the employee's SSN with the appropriate cipher value

Exercise 2: Spring Services Encryption (3)

6) Add a new environment variable to the springline.services project

   Name: SPRINGLINE_SECRET_KEY

   Value: <Enter your secret password>

7) Edit the customer service Employee.java entity

   Change the getSsn() method to decrypt the value

   Change the setSsn() method to encrypt the value

Spring Security | REST Security

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Bean Validation
- REST Authentication
- REST Transport Encryption

Bean Validation

Spring 4.x supports Bean Validation 1.0 (JSR-303) and 1.1 (JSR-349)

Built in validation constraint annotations

Custom validation annotation

REST / Controller valid annotation

Built In Validation Annotations

Bean validation 1.1 (JSR-349) built in constraints:

@Null	@NotNull	@AssertTrue	@AssertFalse
@Min	@Max	@DecimalMin	@DecimalMax
@Size	@Digits	@Past	@Future
@Pattern			

Bean Validation 1.1 Custom Annotation (1)

Example declaring a constraint validator interface called *NameConstraint*:

```
@Target({ElementType.METHOD,ElementType.FIELD,ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=NameValidator.class)
public @interface Name {
   String message() default "{ticket.firstName}"
   Class[] groups() default {};
   Class[] payload() default {};
}
```

Bean Validation 1.1 Custom Validator (1)

Implement a *ConstraintValidator* class:

```
import javax.validation.ConstraintValidator;

public class NameValidator implements ConstraintValidator<Name, String>
{
   public boolean isValid(String object, ConstraintValidatorContext
      constraintContext {

      return NameValidationService.validate(object);
   }
}
```

Bean Validation 1.1 Model Example

Example model object using the constraint validators:

```
public class TicketSearchModel {

   @Name
   private String name;

   @Pattern(regexp = "|(^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$)")
   private String status;

   @Future
   private String date;
}
```

Bean Validation 1.1 REST Controller

Example REST controller enabling bean validation on the TicketSearchModel object:

```
@RequestMapping(path = "/api/ticket/search", method = RequestMethod.POST)
public ResponseEntity<ArrayList<Ticket>> searchTickets(
   @Valid @RequestBody TicketSearchModel search)
{
   …
}
```

Exercise 3: Spring Service Validation (1)

1) Sign in to Springline

   Username: mwaddams, Password: Stapler

2) View the Customer Service > Advanced Search screen

3) Enter the following search terms to test validation:

```
Gibb
Gibb' OR 1=1 -- !
<script>alert(1);</script>
```

Exercise 3: Spring Service Validation (2)

4) Modify the springline services TicketSearchModel.java to use the @pattern annotation to enforce strict input validation:

```
@Pattern(regexp = "|(^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$)")
```

5) Modify the spring services TicketSearchController.java searchTickets method to use the @Valid annotation

6) Test the advance search screen again with the evil payloads

Spring Security | REST Security

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Bean Validation
- REST Authentication
- **REST Transport Security**

REST Transport Security

Security protections provided by enabling Mutual Transport Layer Encryption (MTLS) between a REST service and web server:

Transport Layer Encryption (HTTPS)

Server to server authentication

REST Transport Security | Server Configuration

1) Create Certificate Authority (CA) key and certificate
2) Create server's private key and certificate signed by the CA
3) Import CA and server certificate into Java Keystore (JKS) files
4) Configure Springline Services to require HTTPS, the trust store, and require client authentication

REST Transport Security | Certificate Authority

Create a certificate authority key and certificate using OpenSSL:

```
openssl genrsa -aes256 -out ca.key 2048

openssl req -x509 -new -nodes -subj '/C=US/O=CDD CA/CN=localhost' -key
ca.key -sha256 -days $((10*366)) -out ca.pem
```

NOTE: Reference only, commands are pre-run on the virtual machine already. See ~/certificates/cmd.sh.

REST Transport Security | Server Certificate

Create a server key and certificate using OpenSSL:

```
openssl genrsa -aes256 -out server.key 2048

openssl req -new -sha256 -subj '/C=US/O=Spring Services/CN=localhost' -
key server.key -out server.csr

openssl x509 -req -CA ../ca.pem -CAkey ../ca.key -in server.csr -out
server.cer -days $((10*366)) -CAcreateserial
```

NOTE: Reference only, commands are pre-run on the virtual machine already. See ~/certificates/cmd.sh.

REST Transport Security | Server Keystore

Create a server and ca keystore using OpenSSL and keytool:

```
openssl pkcs12 -export -name server -in server/server.cer -inkey
server/server.key -out server/server.p12

keytool -importkeystore -srckeystore server/server.p12 -destkeystore
server/server.jks

keytool -importcert -file ca.pem -alias ca -keystore ca.jks -noprompt
```

NOTE: Reference only, commands are pre-run on the virtual machine already. See ~/certificates/cmd.sh.

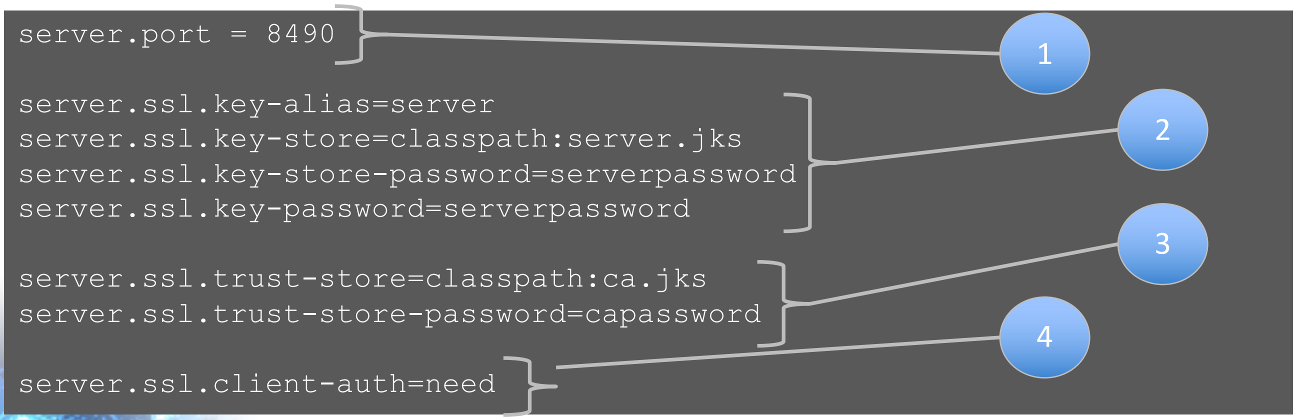
REST Transport Security | REST Configuration

Springline Services application:

1) Configure HTTPS port
2) Configure the server’s private keystore
3) Configure the server’s trust store
4) Require client authentication

Web Services Configuration

Web Services application.properties configuration:



```
server.port = 8490

server.ssl.key-alias=server
server.ssl.key-store=classpath:server.jks
server.ssl.key-store-password=serverpassword
server.ssl.key-password=serverpassword

server.ssl.trust-store=classpath:ca.jks
server.ssl.trust-store-password=capassword

server.ssl.client-auth=need
```

REST Transport Security | Client Configuration

1) Create client's private key and certificate signed by the CA
2) Import CA and client certificate into Java Keystore (JKS) files
3) Configure Springline Web app to send its client certificate to the REST service for authentication and trust the certificate authority

REST Transport Security | Client Certificate

Create a client key and certificate using OpenSSL:

```
openssl genrsa -aes256 -out client.key 2048

openssl req -new -sha256 -subj '/C=US/O=Springline Web/CN=localhost' -
key client.key -out client.csr

openssl x509 -req -CA ../ca.pem -CAkey ../ca.key -in client.csr -out
client.cer -days $((10*366)) -CAcreateserial
```

NOTE: Reference only, commands are pre-run on the virtual machine already. See ~/certificates/cmd.sh.

REST Transport Security | Client Keystore

Create a client keystore using OpenSSL and keytool:

```
openssl pkcs12 -export -name client -in client/client.cer -inkey
client/client.key -out client/client.p12

keytool -importkeystore -srckeystore client/client.p12 -destkeystore
client/client.jks
```

NOTE: Reference only, commands are pre-run on the virtual machine already. See ~/certificates/cmd.sh.

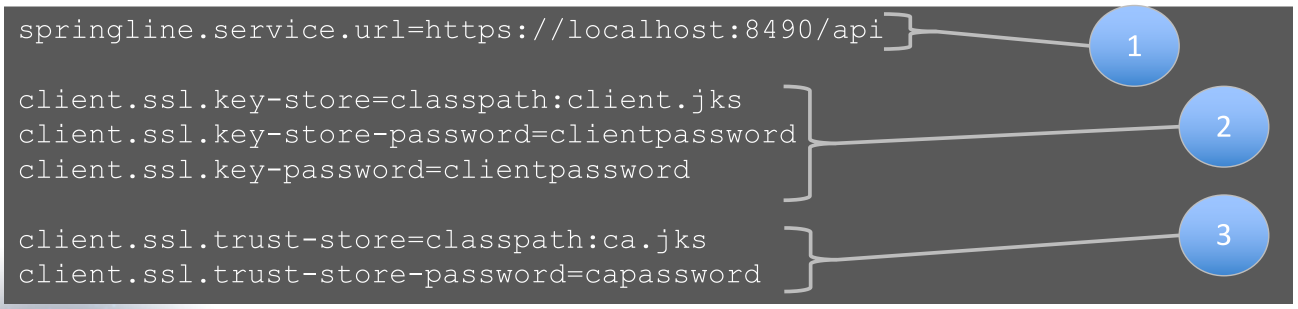
REST Transport Security | Web Configuration

Springline Web application:

1) Configure the new service API URL (protocol and port)
2) Configure the client's private keystore
3) Configure the client's trust store
4) Modify SecurityConfig.java to read the properties and send a custom *SSLContext*

Client Rest Template Properties

Web application.properties configuration:



```
springline.service.url=https://localhost:8490/api

client.ssl.key-store=classpath:client.jks
client.ssl.key-store-password=clientpassword
client.ssl.key-password=clientpassword

client.ssl.trust-store=classpath:ca.jks
client.ssl.trust-store-password=capassword
```

Security Configuration Read Properties

SecurityConfig.java property values:

```
@Value("${client.ssl.trust-store-password}")
private String trustStorePassword;

@Value("${client.ssl.trust-store}")
private Resource trustStore;

@Value("${client.ssl.key-store-password}")
private String keyStorePassword;

@Value("${client.ssl.key-password}")
private String keyPassword;

@Value("${client.ssl.key-store}")
private Resource keyStore;
```

Rest Template Http Client

Client RestTemplate *HttpClient* configuration:

```
private HttpClient httpClient() throws Exception {

   SSLContext sslcontext = SSLContexts.custom()
      .loadTrustMaterial(trustStore.getFile()
         , trustStorePassword.toCharArray())
      .loadKeyMaterial(keyStore.getFile()
         , keyStorePassword.toCharArray(),keyPassword.toCharArray())
   .build();

   SSLConnectionSocketFactory sslConnectionSocketFactory = new
     SSLConnectionSocketFactory(sslcontext, new NoopHostnameVerifier());

   return HttpClients.custom().setSSLSocketFactory(
      sslConnectionSocketFactory).build();
}
```

Spring Multi Factor Authentication

Spring Boot MTLS example:

- Written by Joshua Outwater
- Mutual TLS authentication between 2 web service Spring Boot apps
- https://github.com/joutwate/mtls-springboot

Exercise 4: Spring Service MTLS (1)

The Springline Services application is missing system authentication and transport layer encryption

Your goal is to enable both using client authentication and TLS between the Springline Web and Springline Service applications

Exercise 4: Spring Services MTLS (2)

1) Browse to springline services and ensure you can see the JSON
2) Configure the springline service application.properties:

   Change the port to use 8490

   Set the server.ssl.key-store to server.jks

   Set the server.ssl.trust-store to ca.jks

   Set need client authentication to true

Exercise 4: Spring Services MTLS (3)

3) Browse to springline services over HTTPS on port 8490 and ensure you can see the JSON
4) Enable MTLS

   Set need client authentication to true
5) Browse to springline services over HTTPS on port 8490 again and observer the back certificate handshake error

Exercise 4: Spring Services MTLS (4)

6) Configure the springline web application.properties:

   Add custom properties for the client key and trust store
7) Modify the springline web SecurityConfig.java:

   Add new properties for the client key and trust store values

   Edit the httpClient() method to create a custom SSLContext that uses the client key and trust store

Exercise 4: Spring Services MTLS (5)

8) Browse to springline web again, search for case 42, and verify the case search works as expected

Spring Security | Transport Encryption

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- HTTPS
- Strict Transport Security
- Public Key Pinning

Transport Layer Encryption

Transmitting sensitive information in cleartext

Can lead to the following:

- User account hijacking
- Identify theft
- Loss of confidential information
- Loss of trust in the application's security

Spring Boot HTTPS Configuration

Application.properties configuration:

```
# web server TLS config
server.port = 8443
server.ssl.key-alias=client
server.ssl.key-store=classpath:client.jks
server.ssl.key-store-password=clientpassword
server.ssl.key-password=clientpassword
```

Require Secure Channel

Example require secure channel configuration:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .requiresChannel().anyRequest().requiresSecure();
}
```

HTTP Strict Transport Security (HSTS)

HTTP response header that prevents man-in-the middle attacks

- Requires HTTPS for all future connections to a domain
- Supported by all major browsers

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

Spring Security Configuring HSTS

Spring Security has built in support for sending the HSTS response header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .headers()
         .httpStrictTransportSecurity()
            .includeSubDomains(true)
            .maxAgeInSeconds(31536000)
}
```

HTTP Public Key Pinning (HPKP)

Response header that prevents man-in-the middle attacks

- White list of the server's public keys
- Supported by FF 46+, Chrome 45+, Opera 38+
- Not supported by IE, Edge, or Safari

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

Spring Security Configuring HPKP

Spring Security has built in support for sending the HPKP response header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .headers()
         .httpPublicKeyPinning()
         .maxAgeInSeconds(31536000)
         .reportOnly(false)
         .includeSubDomains(true)
         .reportUri("https://cdd.net/pkp")
         .addSha256Pins("d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM="
            ,"E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=");
}
```

Reading the Public Key Pin

OpenSSL options for extracting the base64 SHA-256 digest from the key file or certificate file:

```
training@ubuntu:~$ openssl rsa -in <keyfile.key> -outform der -pubout |
openssl dgst -sha256 -binary | openssl enc -base64


training@ubuntu:~$ openssl x509 -in <certificate.cer> -pubkey -noout |
openssl rsa -pubin -outform der | openssl dgst -sha256 -binary |
openssl enc -base64
```

Exercise 5: HTTPS Configuration (1)

The Springline application is sending data between the user's browser and web server in cleartext

Your goal is to update the Springline Web application to enable HTTPS and configure the HSTS and HPKP headers

Exercise 5: HTTPS Configuration (1)

Require HTTPS for the Springline web application:

1) Application.properties

   Set the server.port to 8443

   Set the server.ssl properties to use client.jks keystore

2) SecurityConfig.java

   Change the configuration to require secure channel communication

Exercise 5: HTTPS Configuration (2)

Enable the HSTS and HPKP security headers

3) Enable the Strict-Transport-Security header
   - Max-age: 31536000, include subdomains
4) Enable the Public-Key-Pins header
   - Run ~/certificates/keypin.sh to obtain key pin hash
   - Max-age: 31536000, include subdomains
   - Disable report only, add client key pin value

Exercise 5: HTTPS Configuration (3)

5) Browse to the springline web (HTTPS) application and verify it uses the following:
   - https://localhost:8443
   - Use the developer tools to inspect the headers and verify server sends the HSTS and HPKP headers

Spring Security | Authorization

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Authorize Requests
- Role-based Access Control
- Permission-based Access Control

Spring Authorize Request Configuration

Global authorization rules for the entire application:

Define an Ant Matcher that identifies anonymous endpoints

All other requests require an authenticated user

Authorize Request Configuration Example

Example authorize request configuration

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
      .antMatchers("/", "/favicon.ico", "/css/**", "/images/**",
         "/js/**", "/themes/**", "/account/login", "/home",
         "/account/register").permitAll()
      .anyRequest().authenticated()
}
```

Spring Security Granted Authorities

Spring Boot and Spring Security 4 are moving away from role-based access control

Principle objects are now granted a set of permissions, aka authorities

Loading User Authorities

Authorities are read from the user repository:

```
@Service
@Transactional
public class UserService implements UserDetailsService, Serializable {
   @Override
   public UserDetails loadUserByUsername(final String username) {
      final User user = userRepository.findByUserName(username);
      […]
      List<GrantedAuthority> auth =
         AuthorityUtils.commaSeparatedStringToAuthorityList(user.getRoleNames());

      return new org.springframework.security.core.userdetails.User(
         username,  user.getPassword(), auth);
   }
}
```

Thymeleaf Authority Constraint Example

Example using the sec:authorize helper check the hasAuthority permission

templates > fragments > master.html

```
<h3>Human Resources</h3>
<ul>
   <li><a th:href="@{/humanresources/employee}">My Profile</a></li>
   <li sec:authorize="hasAuthority('Manager')">
      <a th:href="@{/humanresources/manage}">Manage Employees</a>
   </li>
</ul>
```

Spring Global Method Security

Enables annotation-based security constraints

- @EnableGlobalMethodSecurity

Annotations can be added to classes or methods to limit access:

- @Secured
- @PreAuthorize

Enabling Global Method Security

SecurityConfig.java

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled =
true)
@Import({customerServicePersistenceConfig.class})
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    //Configure security here
}
```

Role-based PreAuthorize Example

Authorization rule limiting access to users with the "Manager" authority

```
@RequestMapping(value = MANAGE_VIEW_NAME, method = RequestMethod.GET)
@PreAuthorize("hasAuthority('Manager')")
public String getManage(Model model) {
   return MANAGE_VIEW_NAME;
}
```

Permission-based PreAuthorize Example

Combining authority-based permissions with contextual data access

PreAuthorize can evaluate any valid EL expression

```
@PreAuthorize("#employee.name == authentication.name")
public void doEmployeeUpdate(Employee employee) {
    //Update database
}
```

Exercise 6: Springline Authorization (1)

The Springline application does not properly authorize requests to the human resources employee data

Your goal is to restrict access to the manage employee screen to users with the "Manager" authority

Exercise 6: Springline Authorization (2)

Testing Springline for authorization issues:

1) Sign in with username "blumbergh" and password "WorldsBestBoss"
2) Access the Human Resources > Manage Employees screen and copy the URL
3) Sign out and browse to the manage employee screen using the username "mwaddams" and password "Stapler"

Exercise 6: Springline Authorization (3)

Restrict access to the manage employee screen to users with the "Manager" authority:

4) Enable global method security in the SecurityConfig.java file
5) Add the hasAuthority check to the EmployeeController.java getManage method
6) Attempt to access the Manage Employee screen using the username "mwaddams" and password "Stapler"

Spring Security | Session Management

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Session Fixation
- Insecure Logoff

Session Fixation

Permits an attacker to hijack a valid user session.

When authenticating a user, the web application doesn't assign a new session ID, making it possible to use an existing session ID.

Spring Security Session Fixation Protection

Protection is now built into Servlet containers and Spring Security

HTTP Session Fixation Options:

- None
- New Session
- Migrate Session (default for Servlet 3.0 and prior)
- Change Session Id (default for Servlet 3.1+)

Insecure Session Fixation Configuration

Example insecure session management configuration

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .sessionManagement()
      .enableSessionUrlRewriting(true)
      .sessionFixation().none()
      .and()
}
```

Secure Session Fixation Configuration

Example secure session management configuration

Migtate session is the default behavior

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .sessionManagement()
      .enableSessionUrlRewriting(false)
      .sessionFixation().migrateSession()
      .and()
}
```

Insecure Logout

Insecure Logout vulnerabilities occur when an application does not provide users with a way to destroy:

- Session cookies
- Authentication cookies

Insecure Logout Configuration

Example insecure logout configuration

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .logout()
         .logoutRequestMatcher(new
            AntPathRequestMatcher("/account/logout"))
         .permitAll()
         .logoutSuccessUrl("/home")
         .invalidateHttpSession(false)
}
```

Secure Logout Configuration

Example secure logout configuration

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .logout()
         .logoutRequestMatcher(new
            AntPathRequestMatcher("/account/logout"))
         .permitAll()
         .logoutSuccessUrl("/home")
         .invalidateHttpSession(true)
         .deleteCookies("JSESSIONID")
}
```

Exercise 7: Springline Session Management (1)

The Springline application does not properly manage the session id during logon and logout

Your goal is to enable the Spring Security session management protections (session fixation, disable session rewrite, and logout)

Exercise 7: Springline Session Management (2)

Test the Springline application for session fixation and insecure logout vulnerabilities:

1) Use Burp Site to capture traffic
2) Login as “mwaddams” and then logout
3) Inspect the burp traffic and observe the JSESSION id before, during, and after the authenticated session
4) Repeat the process, what did you notice?

Exercise 7: Springline Session Management (3)

Reconfigure Springline to use secure session management:

1) Enable the session fixation protection
2) Disable URL session rewriting
3) Invalidate the session during logout
4) Delete the JSESSION id cookie during logout
5) Retest the login and logout process to verify the fixes

Spring Security | Cross-Site Scripting

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Expression Language Injection
- JSTL Encoding
- Thymeleaf Encoding
- OWASP Java Encoder
- X-XSS-Protection
- Content-Security-Policy

Expression Language Injection

Occurs when attacker controlled data is passed to the EL interpreter

Vulnerable Spring tags include:

<spring:message code="${param.message}" />

<spring:eval expression="${param.expression}" />

Expression Language Injection Example

Example from a JSP file:

```
<spring:eval expression="${param.expression}" />
```

What happens when an attacker enters the following on the URL:

```
Vulnerable.jsp?expression=T(java.lang.Runtime).getRuntime().exec("evil.
exe")
```

Expression Language Injection Defenses

Avoid using vulnerable Spring tags:

<spring:message />

<spring:eval />

If required, validate and encode data to ensure it does not trigger an expression evaluation

Cross-Site Scripting

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper encoding.

- Execute scripts in the victim's browser
- Hijack user sessions
- Deface web sites
- Redirect the user to malicious sites.

Cross-Site Scripting Defenses

Encode all dynamic data written to the browser

- JSTL, Thymeleaf, Java Encoder

Context matters

- HTML, URL, JavaScript, Attribute, CSS, VBScript

Enable browser header protections

- X-XSS-Protections, Content-Security-Policy

JSP Standard Tag Library

JSP Standard Tag Library (JSTL)

- Provides HTML encoding for these characters:

```
  <   >   & "
```

Spring MVC Java Server Pages (JSP) can leverage JSTL to perform limited output encoding

JSP Standard Tag Library Example

Example using c:out to auto-encode dynamic data

```
<div>
   <b>Search Criteria</b>:
   <c:out value="${searchCriteria}"></c:out>
</div>
```

What about other contexts?

- JavaScript, URL, CSS, HTML Attribute

Thymeleaf Output Encoding

Thymeleaf templates auto-encode data written using the th:text attribute:

```
<div>
   <b>Search Criteria</b>:
   <span th:text="${searchCriteria}"></span>
</div>
```

Encoding can be disabled using the th:utext attribute

```
<span th:utext="${searchCriteria}"></span>
```

OWASP Java Encoder

Library that provides output encoding methods to defend against XSS:

https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

Encode.forHtml(input)	Encode.forHtmlAttribute(input)
Encode.forJavaScript(input)	Encode.forUri(input)
Encode.forCssString(input)	Encode.forXmlContent(input)

Java Encoder Example

Thymeleaf templates auto-encode data written using the th:text attribute:

```
<div>
   <b>Search Criteria</b>:
   <span th:utext=”
      ${ T(org.owasp.encoder.Encode).forHtml(ticket.description)}" >
   </span>
</div>
```

X-XSS-Protection Header

HTTP response header that provides defense in depth prevents basic reflected XSS attacks

- Supported by all major browsers
- 0: Domain cannot be framed
- 1: Modifies page stripping payload from the response
- 1; mode=block: Blocks the response and renders a blank page

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

Spring Security Configuring X-XSS-Protection

Spring Security automatically sends the X-XSS-Protection header

Example of explicitly setting the header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .headers()
         .xssProtection().block(true);}
```

Content Security Policy Header

HTTP response header that helps prevents XSS attacks

- 1.0 is supported by all major browsers
- 2.0 is now officially supported by all major browsers (as of Edge 15)

https://www.w3.org/TR/CSP/

Content Security Policy Defenses

CSP is a browser-based specification that allows a web server to:

- Set a whitelist of external resources permitted to be downloaded and executed
- Disable inline JavaScript and CSS styles
- Disable dynamic code execute (eval, setTimeout, etc.)

Content Security Policy 1.0 Keywords

‘self’ - Allow resources from the same origin

‘none’ - Deny all resources

‘unsafe-inline’ – Allow inline resources

‘unsafe-eval’ – Allow dynamic code execution

‘data:’ – Allows data URIs

Content Security Policy 1.0 Directives

default-src	media-src
script-src	frame-src
object-src	font-src
style-src	connect-src
img-src	report-uri

Content Security Policy 1.0 Example

Example from https://mobile.twitter.com:

```
Content-Security-Policy:
    default-src 'self’;
    font-src    'self';
    frame-src   https://*.twitter.com;
    img-src     https://*.twitter.com
                https://*.twimg.com
                https://maps.google.com data:;
    script-src  https://*.twitter.com
                https://*.twimg.com
                https://api-secure.recaptcha.net
                'unsafe-inline' 'unsafe-eval';
    style-src   https://*.twitter.com
                https://*.twimg.com
                https://api-secure.recaptcha.net
                'unsafe-inline';
    report-uri  https://twitter.com/scribes/csp_report;
```

Content Security Policy Violations

Violations captured in the Chrome console:

Content Security Policy Reporting

JSON data sent to the report-uri endpoint:

```
{
    "csp-report":
    {
        "document-uri": "http://demo.cdd.org/home",
        "referrer": "http://demo.cdd.org/index.html",
        "blocked-uri": "http://cddexploit.net/evil.js",
        "violated-directive": "script-src 'self' https://apis.google.com",
        "original-policy": "script-src 'self' https://apis.google.com;
          report-uri http://demo.cdd.org/cspreport"
    }
}
```

Configuring Content-Security-Policy

Spring Security supports the Content-Security-Policy header

Example setting the header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .headers()
         .contentSecurityPolicy("script-src 'self'
            https://trustedscripts.example.com; object-src
            https://trustedplugins.example.com;
            report-uri /csp-report-endpoint/");
}
```

Content Security Policy Testing Tools

csp-tester

- Chrome extension to build and test CSP
- https://github.com/oxdef/csp-tester

CSPTools

- Python based CSP Proxy, Browser, and Parser
- Released by @kennysan at DEFCON 2013
- https://github.com/Kennysan/CSPTools

Exercise 8: Springline XSS (1)

The Springline application does not properly encode dynamic data before rending it in the browser

Your goal is to fix the XSS issues using output encoding and the XSS protection and CSP security headers

Exercise 8: Springline XSS (2)

Test Springline for XSS issues:

1) Login as “mwaddams” and “Stapler”
2) Using Firefox, use the case search screen to exploit persisted XSS:

   Edit case # 42

   Store some evil JavaScript in the Description and Comments fields

   View case #42 again and verify the XSS vulnerabilities exist

Exercise 8: Springline XSS (3)

Fix the Springline XSS issues:

3) Fix the "Description" persisted XSS issue using output encoding

   Edit the search_simple.html template and use the Thymeleaf default encoding or the Java Encoder

4) Fix the persisted XSS issue by setting the CSP and X-XSS-Protection headers. Here is a sample CSP to use:

```
default-src 'self'; style-src 'self' 'unsafe-inline';
```

Exercise 8: Springline XSS (4)

Verify the Springline XSS issues no longer exist:

5) Search for case 42

6) View the case details and confirm the fixes prevent the evil scripts from running

7) Inspect the CSP error in the Firefox console

Spring Security | Cross-Site Request Forgery

- Spring Boot / Security
- Authentication
- Cryptography
- REST Security
- Transport Encryption
- Authorization
- Session Management
- Cross-Site Scripting
- **Cross-Site Request Forgery**
- Response Headers

- CSRF Overview
- CSRF Configuration
- Spring JSP Tag Library
- Thymeleaf

Cross-Site Request Forgery

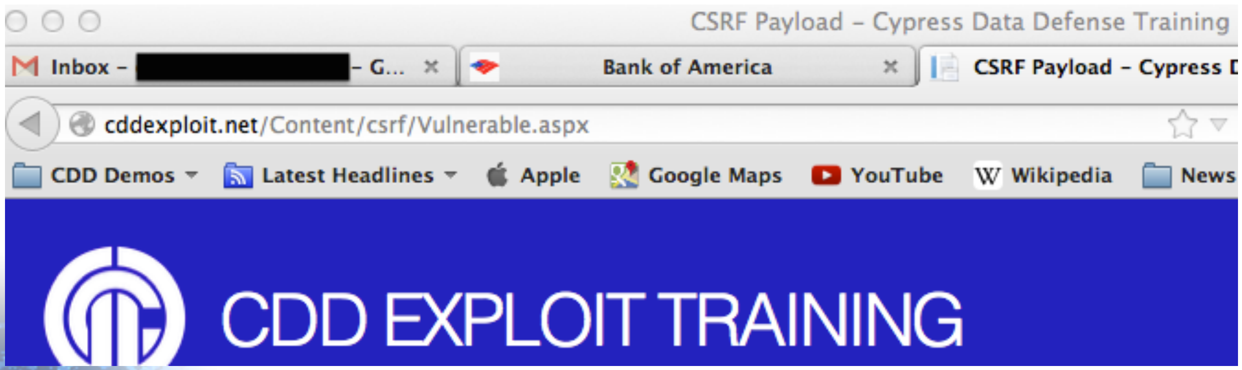
Forces an authenticated victim's browser to send a forged HTTP request

Applications using authentication cookies are inherently vulnerable:

- Audit logs will show the user made the transaction
- User has no knowledge of the transaction

Browser Shared Cookie Collection

Shared cookie collection across browser tabs:



CSRF Payload

Example of the attackers evil web page

```
<form id="csrfForm"
action="http://localhost:8080/account/changepassword" method="POST” >
    <input type="hidden" name="newPassword" value="StorageRoomB" />
    <input type="hidden" name="confirmPassword" value="StorageRoomB" />
</form>
```

CSRF Generated Request

Example of the request generated when the victim visits the evil web page

```
POST /account/changepassword HTTP/1.1
Host: localhost:8080
Cookie: JSESSIONID=2E7F523BE6E086F5EEB593B2B69842D2
Content-Type: application/x-www-form-urlencoded
Content-Length: 53

newPassword=StorageRoomB&confirmPassword=StorageRoomB
```

CSRF Generated Response

Example of the response generated when the victim visits the evil web page

```
HTTP/1.1 200 OK

<div class="alert alert-dismissable alert-success">
   <span>Your password was successfully changed.</span>
</div>
```

Cross-Site Request Forgery Protections (1)

Defending against CSRF attacks requires a value in the request to change for each:

- Session (common solution)
- Request

Commonly achieved using:

- Anti-csrf tokens
- Nonce

Cross-Site Request Forgery Protections (2)

Anti-csrf tokens must be protected for the solution to work:

- Do not send over HTTP connections
- Avoid making data modifications on GET requests
- No XSS vulnerabilities!

Spring Security CSRF Protection

Spring CSRF protection is enabled by default

INSECURE example disabling this feature. Do NOT do this:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .csrf().disable();
}
```

Spring JSP Tag Library

Injecting a CSRF token into a form using Spring's JSP tag library:

```
<form action="/account/manage" method="POST" >
    <input type="password" name="newPassword" />
    <input type="password" name="confirmPassword" />
    <input type="hidden" name="${_csrf.parameterName}"
      value="${_csrf.token}"/>
</form>
```

Protecting AJAX and JSON Requests

Injecting a CSRF token into a JSON request using meta tags:

```
<head>
   <meta name="_csrf" content="${_csrf.token}"/>
   <meta name="_csrf_header" content="${_csrf.headerName}"/>
   <!-- ... -->
</head>
```

Including the token in all AJAX requests that modify data:

```
$(function () {
   var token = $("meta[name='_csrf']").attr("content");
   var header = $("meta[name='_csrf_header']").attr("content");
   $(document).ajaxSend(function(e, xhr, options) {
      xhr.setRequestHeader(header, token);
}); });
```

Thymeleaf Template

Thymeleaf 2.1+ automatically injects a token into form elements when @EnableWebSecurity is set

Example Thymeleaf template:

```
<form method="post" th:action="@{/account/manage}">
   <input id="newPassword" type="password" />
   <input id="confirmPassword" type="password" />
   <input type="submit" value="Change Password" id=”changePassword" />
</form>
```

Thymeleaf HTML Source

On the client, viewing the HTML source shows the _csrf hidden input injected into the page:

```
<form method="post" th:action="@{/account/manage}">
   <input id="newPassword" type="password" />
   <input id="confirmPassword" type="password" />
   <input type="submit" value="Change Password" id=”changePassword" />
   <input type="hidden" name="_csrf"
      value="a47b9e4b-4d49-401e-aa23-fd1b44365d6f" />
</form>
```

Exercise 9: Springline CSRF (1)

The Springline application does not enable CSRF protection

Your goal is to use the CSRF attack payload to exploit CSRF, and then enable Spring Security’s CSRF protection

Exercise 9: Springline CSRF (2)

Test Springline for CSRF:

1) Copy the password hash associated with “mwaddams” in the customer_service user table into a new query
2) Sign in with username “mwaddams” and password “Stapler”
3) Press the CSRF Attack link to execute a CSRF attack
4) Verify the attack worked by looking in the customer_service user table again. Did Milton’s password hash change?
5) Change the password for ”mwaddams” back to the original hash value

Exercise 9: Springline CSRF (3)

Protect Springline from CSRF attacks:

6) Modify the Springline SecurityConfig.java class to enable CSRF protection
7) Login as “mwaddams” again and click the CSRF attack link
8) Verify that the attack failed this time and the password hash remains the same

Spring Security | Response Headers

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers

- Frame Options
- Content Type Options

X-FRAME-OPTIONS Header

HTTP response header that prevents Clickjacking attacks

Tells the browser not to load a site via iframe source

Supported by all major browsers:

DENY: Domain cannot be framed

SAMEORIGIN: Limits framing to same origin policy

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

Spring Security Configuring X-FRAME-OPTIONS

Spring Security automatically sends the X-FRAME-OPTIONS header set to DENY

Example of explicitly setting the header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .headers()
         .frameOptions()
         .sameOrigin();
}
```

Content Type Options Header

HTTP response header that prevents browsers from guessing the response content type

- E.g. JavaScript missing a content type will not be automatically executed

Supported by all major browsers

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

Spring Security Configuring Content Type Options

Spring Security automatically sends the X-Content-Type-Options header

Example of explicitly setting the header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .headers()
         .contentTypeOptions();
}
```

Exercise 10: Springline Clickjacking (1)

The Springline application does not enable the Clickjacking and Content Type Options headers

Your goal is to test Springline Web for Clickjacking, and then enable the Clickjacking and Content Type Options headers

Exercise 10: Springline Clickjacking (2)

Test Springline for Clickjacking:

1) Sign in with username “mwaddams” and password “Stapler”
2) Press the Clickjacking link to test for this issue
3) Modify Springline’s SecurityConfig.java to send the frame options and content type option headers
4) Test the DENY and SAMEORIGIN options. Why do these respond differently?

Spring Boot & Security

Spring Boot / Security
Authentication
Cryptography
REST Security
Transport Encryption
Authorization
Session Management
Cross-Site Scripting
Cross-Site Request Forgery
Response Headers