Application Security Quick Solution Guide

Table of Contents

Introduction

Mission

The purpose of this guide is to offer quick solutions for common application security issues for all applications based on the OASP platform. It's often the case that we need our systems to comply to certain sets of security requirements and standards. Each of these requirements needs to be understood, addressed and converted to code or project activity. We want this guide to prevent the wheel from being reinvented over and over again and to give clear hints and solutions to common security problems.

Is this guide for me?

All presented examples are based on the OASP Java and JavaScript platform. Projects using this platform can benefit and accelerate their activity by using this guide the most. Projects not using the OASP platform, but relying on the Spring framework, can still find plenty of working examples. Projects not using the Spring framework or Java language might still find it interesting to see what security requirements and standards are about.

NOTE

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

We use the expression "secure by design" to indicate, that the solution to the requirement is covered by the OASP platform or language itself. It usually never means (unless stated otherwise), that the platform/language guarantees the requirement to be always satisfied. The developer can try to fight the frameworks or bypass the language features. "Secure by design" means, that the natural solution given by the environment mitigates the security threat.

Content

The entry point to this guide is based on the OWASP Application Security Verification Standard and the OWASP Top 10 list, as the compliance with these is a common project requirement we face today.

- OWASP Application Security Verification Standard (ASVS) v3.0.1
- OWASP Top 10 2017

Please note, although we do support OWASP Top 10, we do not consider it to be the right tool for application security assurance. To name only one reason: no list limited to n items can be used as a base for a comprehensive security assurance plan. If in doubt, please refer to the OWASP ASVS standard instead.

---

1. S1 Basic Architecture and Design

Architecture and design plays a role as part of the Security Assurance strategy.

1.1 S1-1

All application components are identified and are known to be needed.

Purpose: Every library you add to the project raises the attack surface and may impact your security posture. It's not only true for framework libraries like Struts 2 (e.g. CVE-2017-5638) or Spring MVC (e.g. CVE-2014-0054). It's also true for util libraries like Apache Commons (e.g. CVE-2015-7501) and others.

Solution: Do not include any more dependencies to your project then actually needed. Always document why you need a certain library in the project, so you can remove the dependency if functionality gets deleted.

Find a solution that works best for you like writing comments in a pom.xml file, using Excel or documenting in a tool of your choice.

2. S2 Basic Authentication Security

Authentication answers the question, weather "user is who he claims to be". Focus of this solution is the implementation

2.1 S2-1

All pages, functions and resources that require authentication - enforce authentication.

Purpose: Misconfiguration around authentication controls may result in confidential data being exposed to anonymous users.

Solution (Java-based): Create a configuration class that inherits from `WebSecurityConfigurerAdapter` and overrides the `configure(HttpSecurity)` method. Following example is taken from the OASP sample application (class `io.oasp.application.mtsj.general.service.impl.config.BaseWebSecurityConfig`:

```
String[] unsecuredResources = new String[] { "/login", "/security/*", "/services/rest/login",
    "/services/rest/logout", "/services/rest/dishmanagement/*", "/services/rest/imagemanagement/*",
    "/services/rest/ordermanagement/v1/order", "/services/rest/bookingmanagement/v1/booking",
    "/services/rest/bookingmanagement/v1/booking/cancel/*",
    "/services/rest/bookingmanagement/v1/invitedguest/accept/*",
    "/services/rest/bookingmanagement/v1/invitedguest/decline/*",
    "/services/rest/ordermanagement/v1/order/cancelorder/*"};
```

```
http.antMatchers(unsecuredResources).permitAll().anyRequest().authenticated();
```

This approach is a nice example how to implement authentication to be *secure by default*. Every new functionality will be automatically protected unless an explicit exception is added to `unsecuredResources` variable.

of secure authentication functions. **Solution (xml based)**: In Java applications relying on the Spring Security framework (like OASP does) the need for authentication can be configured in the file `beans-security.xml`. Following example is taken from the OASP sample application.

```
<bean id="FilterSecurityInterceptor"
 class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="AuthenticationManager"/>
  <property name="accessDecisionManager" ref="FilterAccessDecisionManager"/>
  <property name="securityMetadataSource">
    <security:filter-security-metadata-source use-expressions="true">
      <security:intercept-url pattern="/" access="isAnonymous()"/>
      <security:intercept-url pattern="/index.jsp" access="isAnonymous()"/>
      <security:intercept-url pattern="/security/login" access="isAnonymous()"/>
      <security:intercept-url pattern="/j_spring_security_login" access="isAnonymous()"/>
      <security:intercept-url pattern="/j_spring_security_logout" access="isAnonymous()"/>
      <security:intercept-url pattern="/services/rest/security/v1/currentuser/" access="isAnonymous() or
 isAuthenticated()"/>
      <security:intercept-url pattern="/" access="isAuthenticated()"/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

In the example we use expression language `use-expressions="true"` like `isAuthenticated()` or `isAnonymous()` to decide, weather the resource identified by the given pattern requires authentication or not.

The last line of the filter configuration defines a pattern / catching all remaining resources and enforcing authentication. This **secure default** ensures all new pages and resources created by the developers are protected, and all exceptions to this rule must be explicitly added to the configuration.

2.2 S2-2

All user passwords are protected with a strong one way hashing function which incorporates salt and a properly configured work factor.

Purpose: Holding passwords for user accounts in plaintext creates many problems with trust (to e.g. system administrators, developers), backups (data can leak), and raises the criticality of every possible data breach. Passwords need to be hashed with a strong key derivation function like bcrypt, scrypt or PBKDF2 which automatically will include salt. A work factor needs to be calculated and properly set as well, as it mitigates some threat of brute force attacks.

Solution: Spring Security offers its own bcrypt implementation to deal with password hashing issues. An example configuration (which uses in-memory authentication in this case) can look like this:

```
auth.inMemoryAuthentication().passwordEncoder(new BCryptPasswordEncoder(10))
    .withUser("waiter").password("$2a
$10$n5UZipMzWmYB18AWbyZES.t9FDJP/9OFp8hKI1rzol6gn/2hVZYE2").roles("Waiter")
    and().withUser("user0").password("$2a
$10$atp6JC.anPKGhNwc1dEK0OQFzPr3/5moIm7bF1i2yBHXFKpA7YMb2").roles("Customer");
```

NOTE:

- The work factor (passed as a parameter to `BCryptPasswordEncoder`, which equals 10 in our example) defines the amount of work (as result time) needed to calculate the digest. Incrementing the value by one means multiplication of the work amount by two.
- Work factor = 10 means calculation of the hash lasting about 80ms on a Lenovo P50 machine (which should be good enough for the purpose of online user authentication). Work factor = 11 would result in the calculation lasting about 170ms on same machine.
- To high work factors will raise the threat of DOS attacks on the authentication functionality. Too low work factor will not offer enough protection.
- For the purpose of storing hashes in the system they need to be calculated first. Following code can help to do that: `BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(10); String digest = encoder.encode("put the password here"); System.out.println(digest);`
- Although the passwords are hashed with a one-way function, it's still not allowed to expose them in logs, APIs or forms. Digests will only slow down an attack to crack the hashes, not prevent it.

2.3 S2-3

All authentication controls are enforced on the server side.

Purpose: An attacker has the ability to bypass any security control on the client side. Because of this it is crucial to implement authentication controls on the server side.

Solution: Secure by design. In the OASP architecture it is always the server side which is responsible for the authentication. The developers must not bypass the authentication functions in any way.

2.4 S2-4

All authentication controls fail securely.

Purpose: There are many things that can go wrong around authentication from the business or technical perspective. It is crucial that authentications never **fail open** (which is the opposite from **fail secure** - users get access to the application despite authentication failure). It is also necessary to prevent any exposure of stack traces (reveals internal system information to the attacker; → TODO) or username enumeration (→ S2-7) possibilities.

Solution: Spring Security handles the technical security aspects of this solution well (prevents timing attacks to enumerate usernames, prevents SQLi, and other). The developer must just handle the displaying of the error message properly.

2.5 S2-5

Credentials, and all other identity information handled by the application(s), do not traverse unencrypted or weakly encrypted links.

Purpose: TCP/IP networks are insecure by default. Any kind of information that travels over insecure channel (HTTP instead of HTTPS) may be revealed to local or (sometimes) remote attackers without the user notice.

Solution: The application on the integration and production environments must always protect the connection with TLS (→ TODO). Ideally the unencrypted HTTP protocol is turned off. If for some reason the unsafe HTTP connection is needed, the server must enforce the authentication dialog and all pages within the session to be served only via TLS. This can be achieved by setting the attribute `requires-channel="https"` of the element `security:intercept-url`. In the Java-based Spring Security configuration the relevant part of code looks like this: `http.requiresChannel().antMatchers("/*").requiresSecure()` For more information see this tutorial.

All sensitive information (including credentials) must only be passed in the HTTP message body, never in the URL. This prevents the sensitive information from being stored in browsers history or in logs on the web server.

2.6 S2-6

The forgotten password function and other recovery paths do not reveal the current password and the new password is not sent in clear text to the user.

Purpose: Passwords are very sensitive. If the user can recover his password from the system, the attacker will be able to recover users password as well.

Solution: Passwords need to be protected using a one-way hashing function (→ S2-2). This way, once they are set, they can not be revealed any more.

2.7 S2-7

Username enumeration is not possible via login, password reset, or forgot account functionality.

Purpose: If the the system is protected by username and passwords, the attacker must know both to gain access. Although username is not sensitive data, it still should not be revealed over the app functionality, as this would unnecessarily support the attacker.

Solution: Prevention of username enumeration can be a tricky thing to do. Not only must the application respond to login attempts in a manner that does not reveal the information, whether the user exists or not. It must also prevent timing attacks that can reveal the information in an indirect way (see this). Note following:

- User authentication in the OASP applications is handled by Spring Security. We expect Spring to do it right.
- Functionality like password reset or password change needs still to be implemented by the developers in a correct manner that does not leak information to the attacker.
- It can be a challenge to implement the functionality of creating a new user account in a way that does not allow username enumeration (the app usually asks for the username in the very first dialogue window and then automatically reports whether this name is free or taken). One of the ways to tackle this problem is to:

  1. Ask for username (usually a mail).
  2. Send a mail to the user with the link to a page to continue the registration process OR send a mail stating that the account already exists.

2.8 S2-8

There are no default passwords in use for the application framework or any components used by the application.

Purpose: The attacker can try to access the system by brute forcing user account credentials. But he can also try to access the database (or other services) directly by brute-forcing the DB user credentials.

Solution: This is not an issue one can solve with technology or frameworks. Treat is as a non-functional requirement in the development and deployment process not to use passwords like `admin`, `password`, `12345` or any password from e.g. Adobe top 100 passwords list.

2.9 S2-9

Password entry fields allow, or encourage, the use of passphrases, and do not prevent password managers, long passphrases or highly complex passwords being entered.

Purpose: For some reason sometimes systems prevent users from entering passwords longer than 8 characters. This weakens the systems protection and opens door to brute force attacks. Other reasons for allowing long passphrases document this xkcd cartoon below: ! XKCD passwords

Solution: Spring Security will not prevent the usage of password managers or long passphrases. This is a nice secure default.

2.10 S2-10

Account identity authentication functions (such as update profile, forgot password, disabled / lost token, help desk or IVR) that might regain access to the account are at least as resistant to attack as the primary authentication mechanism.

Purpose: Rather obvious: The chain is as strong as the weakest link. The attacker will choose the easiest path to gain access to the system.

Solution: Requirement not supported by the OASP platform, as it addresses topics of business processes and functionality of the system. Treat this as a non-functional requirement to be included in the applications design phase.

2.11 S2-11

The changing password functionality includes the old password, the new password, and a password confirmation.

Purpose: Some critical system functionality may re-authenticate the user before performing the task. This is done to prevent that someone changes the users password by gaining physical access to his machine in the short moment the user is away or distracted.

Asking the user for the password twice is done to prevent the user from losing access to the system because of a misspelled password or passphrase.

Solution: Spring Security does not support this functionality. It needs to be implemented by the developer this way.

2.12 S2-12

Anti-automation is in place to prevent breached credential testing, brute forcing, and account lockout attacks.

Purpose: Even without any knowledge of the system, attackers may try to brute force the users credentials. On the other side, even without knowing the users credentials, the attacker can try to lock out valid users from accessing the system by triggering account lockouts due to brute forcing.

Solution: Protection from brute force and DOS attacks is one of the hardest problems in Application Security. There is a big gap between how easy it is for the attacker to run such attacks, and how hard it is for a developer to implement proper security controls.

This issue can be handled in many different ways. One of the common approaches is to use a Web Application Firewall (WAF) and configure it properly. But some security controls can also be implement by using Spring Security features.

2.13 S2-13

Forgotten password and other recovery paths use a TOTP or other soft token, mobile push, or other offline recovery mechanism. Use of a random value in an e-mail or SMS should be a last resort and is known weak.

Purpose: If your password recovery mechanism is weaker then your authentication, then the attacker will use this way to gain access to your system.

Solution: Requirement not supported by the OASP platform, as it addresses topics of business processes and functionality of the system. Treat this as a non-functional requirement to be included in the applications design phase.

2.14 S2-14

If shared knowledge based questions (also known as "secret questions") are required, the questions do not violate privacy laws and are sufficiently strong to protect accounts from malicious recovery.

Purpose: If your password recovery mechanism is weaker then your authentication, then the attacker will use this way to gain access to your system.

Solution: Requirement not supported by the OASP platform, as it addresses topics of business processes and functionality of the system. Treat this as a non-functional requirement to be included in the applications design phase.

2.15 S2-15

Verify that measures are in place to block the use of commonly chosen passwords and weak passphrases.

Purpose: If you allow the users of your application to choose any password they want, you will end up with ~2% of all system accounts protected by passwords like `password`, `12345` or `abc123`. (Read more)

Solution: There are many solutions how to tackle this problem. This article offers one of them.

3. S3 Basic Session Protection

The session of the application users lies in the centre of the applications security and needs to be as tightly protected as the users credentials within the application.

NOTE

This chapter is about the session protection. Some modern approaches for web applications we see today use no session, are completely stateless and rely tokens like JWT. For those application the contents of this chapter are in big part irrelevant.

3.1 S3-1

Containers default session management control implementation is used by the application.

Purpose: Don't design and implement session management yourself. We've seen it many times. People end up repeating same security issues over and over again. And proper session management is less trivial than it looks.

Solution: Secure by design. The OASP platform and Spring Security relies on the JEEs default session management implementation (for all web applications and API that are not stateless/session-less).

3.2 S3-2

The session is invalidated when the user logs out.

Purpose: After the user logs out of the application, we want his session to be irrecoverably destroyed.

Solution: Secure default, if the Spring Security logout functionality is triggered correctly.

3.3 S3-3

The session times out after a specified period of inactivity.

Purpose: Infinite lasting session raises the attack surface of the application.

Solution: Secure default of the Tomcat container. The default definition of how long the session is active is configured under `TOMCAT_HOME/conf/web.xml` (can be overridden with the web.xml file under `WebApplication/webapp/WEB-INF/web.xml`) and looks like:

```
<!-- ==================== Default Session Configuration ================ -->
<!-- You can set the default session timeout (in minutes) for all newly   -->
<!-- created sessions by modifying the value below.                       -->
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

The value can also be set over Spring Boots properties file with the property `server.session.timeout`.

The session timeout must always be set to a reasonable amount of time and never turned off. The 'remember me' functionality is considered less secure and shouldn't be implemented without a good reason. In case the 'remember me' functionality is needed, the Spring Security remember me pattern should be used.

3.4 S3-4

All pages that require authentication to access them have logout links.

Purpose: We want the user to be able to close his session when he's done with the work, so the attacker, who might gain physical access to the system, can not use his session any more.

Solution: Consider this requirement in the application design phase.

3.5 S3-5

The session id is never disclosed in URLs, error messages, or logs. This includes verifying that the application does not support URL rewriting of session cookies.

Purpose: Session ids are highly sensitive, as they allow to access the system in a similar fashion the credentials do. They must not be disclosed in URLs (sometimes the system administrators need to log them out), error messages (they land in logs) or logs themself.

Solution: This is a secure default of the OASP platform, yet still this can be broken by the developers actions. The developer must ideally never access the sessions id (`HttpSession.getId()`) for any reason. In cases where the application stores all requests content in a log file, the logs must exclude the session cookie.

3.6 S3-6

Successful authentication and re-authentication generates a new session and session id.

Purpose: This is needed to prevent the threat of session fixation.

Solution: Secure default of Spring Security.

3.7 S3-7

Session ids stored in cookies have their path set to an appropriately restrictive value for the application, and authentication session tokens additionally set the "HttpOnly" and "secure" attributes.

Purpose: Session ids are set over `Set-Cookie` headers in HTTP responses like `Set-Cookie: JSESSIONID=AVASMKLAJWJV34mlkjsadflKJMSLMKJ234MLKJFSALK; path=/app1; HttpOnly; secure`.

The `path` attribute is important, if more than one application is running on the server. We don't want our session cookies to be sent to applications different then ours. The flag `HttpOnly` prohibits JavaScript from accessing the cookies value (reduces the impact of XSS vulnerabilities). The flag `secure` allows the cookie to be sent only over TLS protected channels.

Solution: The fastest way for Spring Boot applications to change cookie parameters is through the application properties file:

```
server.session.cookie.domain= = Domain for the session cookie.
server.session.cookie.http-only= = "HttpOnly" flag for the session cookie. (true (default) / false)
server.session.cookie.max-age= = Maximum age of the session cookie in seconds.
server.session.cookie.name= = Session cookie name.
server.session.cookie.path= = Path of the session cookie.
server.session.cookie.secure= = "Secure" flag for the session cookie. (true / false)
```

3.8 S3-8

The application limits the number of active concurrent sessions.

Purpose: We want to prevent single user from being able to run a DOS attack on the session manager and exhaust its resources.

Solution: Spring security can limit the maximum amount of concurrent user sessions. Create a configuration class that inherits from `WebSecurityConfigurerAdapter` and overrides the `configure(HttpSecurity)` method. Following code allows only one user session to be opened and redirects the user to URL if he authenticates more than once: `http.sessionManagement().maximumSessions(1).expiredUrl("URL")`.

3.9 S3-9

An active session list is displayed in the account profile or similar of each user. The user should be able to terminate any active session.

Purpose: You go to the library, log into your account on one of the machines there, but forgets to log out. You want to be able to close the session remotely when you return back home.

Solution:

TODO: A nice solution could result from the combination of these two articles. I don't really like how the session ids are exposed in the second article. This kind of data should not be visible on the UI like this. But if we store a bit more metadata in the session (like in the first article), we might be able to identify the session with more human-like factors. This looks rather complicated though.

3.10 S3-10

User is prompted with the option to terminate all other active sessions after a successful change password process.

Purpose: If you suspect your account to be compromised, you want to have the ability to regain the ownership.

Solution: Same as above.

4. S4 Basic Access Control

This package describes basic security access controls.

4.1 S4-1

The principle of least privilege exists and is properly implemented in the application.

Purpose: Least privilege means, that the user has the minimum access rights to applications functionality that is needed for him to perform his tasks. This is one of the core principles of Application Security. Users with to big access rights might use them to influence the confidentiality, integrity or availability of the system.

Solution: This is a business requirement in the first place and has little to do with the technical platform the application is using. We need to identify all user roles that deal with the application, understand their work and the scope of their competences, and then implement security controls in the application as described with solution S4-2.

4.2 S4-2

Users should only be able to access functions, data files, URLs, controllers, services, and other resources, for which they possess specific authorization.

The access to the application functions is controlled by a group of roles with specific permissions. All groups and permissions are defined in one XML file which example looks like below:

```
<?xml version="1.0" encoding="UTF-8"?>
<access-control-schema>
    <group id="ReadMasterData" type="group">
       <permissions>
          <permission id="FindOffer"/>
       </permissions>
    </group>
    <group id="Cook" type="role">
      <inherits>
        <group-ref>ReadMasterData</group-ref>
      </inherits>
      <permissions>
        <permission id="SaveOffer"/>
      </permissions>
    </group>
</access-control-schema>
```

The usual place to store this definition is in the src/main/resources directory under `/config/app/security/`. To make the schema definition visible to Spring, following bean definitions must be included:

```
<bean id="AccessControlProvider"
 class="io.oasp.module.security.common.impl.accesscontrol.AccessControlProviderImpl"/>
<bean id="AccessControlSchemaProvider"
 class="io.oasp.module.security.common.impl.accesscontrol.AccessControlSchemaProviderImpl"/>
```

Based on the definition above users access rights (meaning belonging to a group containing a defined permission) can be enforced by using annotation `@RolesAllowed` as as in the example below:

```
@RolesAllowed(PermissionConstants.FIND_OFFER)
public List<OfferEto> findAllOffers() {
  return getBeanMapper().mapList(getOfferDao().findAll(), OfferEto.class);
```

```
}
```

If the user accessing the `findAllOffers` method belongs to a group which contains the permission `PermissionConstants.FIND_OFFER`, the method will be executed, else an exception will be thrown.

Additionally annotations like `@PermitAll` and `@DenyAll` can be used, to allow or deny all user access to a function.

4.3 S4-3

Access to sensitive records is protected, such that only authorized objects or data is accessible to each user (for example, protect against users tampering with a parameter to see or alter another user's account).

Purpose: Access control must be made on two levels: functional (as described in S4-2) and data (S4-3). The user might be able to e.g. edit blog data, but he still must not be able to edit data he's not the owner of. This problem is closely related to the problem of insecure direct object references. Refer to the (OWASP pages) for more information.

Solution: This requirement is not addressed by the OASP platform (yet). In case a need exists to protect direct object references in the system (different objects are accessible for different user groups), it needs to be custom implemented and tested.

> **Note**
>
> The proper solution of the problem of insecure direct object references are secure direct object references (access control rights are checked), NOT indirect object references (parameters contain some kind of a secret) as some pages might suggest.

4.4 S4-4

Directory browsing is disabled unless deliberately desired.

Purpose: Directory browsing allows for listing of directories and directory contents that reside on the server. If not deliberately needed, this might be very interesting for the attacker, because: - it reveals information about internal directory and file structure on the server, - it can reveal sensitive information (database backups, .git folders, …).

Solution: Secure default of the Tomcat server. The `DefaultServlet` configuration (Tomcats `/conf/web.xml` file), which can be used to list directory contents, has the parameter `listings` set to `false`. Let's keep it this way.

4.5 S4-5

Access controls fail securely.

Purpose: The opposite of fail securely - fail open - means, that the attacker can gain something out of an access control failure, be it: - access to the system resources, - stack trace revealing technical system information, - other.

Solution: We expect the OASP access control mechanisms as described in S4-1 together with S6.1 (system error handling) to allow applications to fail securely.

4.6 S4-6

The same access control rules implied by the presentation layer are enforced on the server side.

Purpose: Every client side control can be bypassed by the attacker. In the end only server side controls can prevent the attack.

Solution: This must be properly implemented and tested by the developer team.

4.7 S4-7

The application or framework generates strong random anti-CSRF tokens unique to the user as part of all high value transactions or accessing sensitive data, and that the application verifies the presence of this token with the proper value for the current user when processing these requests.

Purpose: Cross-site Request Forgery is an attack, that forces the end user to perform unwanted actions on his behalf (see OWASP).

Solution: CRFS tokens are generated using Spring HttpSessionCsrfTokenRepository. They are loaded from the HttpServletRequest and stored in the HttpSession. The Spring implementation generates strong random tokens → `UUID.randomUUID().toString()`.

Spring configuration for Cross Site Request Forgery is stored in beans-security-filters.xml:

```
<bean id="HttpSessionCsrfTokenRepository"
 class="org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository"/>
<bean id="CsrfFilterWrapper" class="io.oasp.module.web.common.base.ToggleFilterWrapper">
   <property name="delegateFilter" ref="CsrfFilter"/>
   <property name="enabled" value="${oasp.filter.csrf}"/>
</bean>
<bean id="CsrfFilter" class="org.springframework.security.web.csrf.CsrfFilter">
   <constructor-arg>
       <ref bean="HttpSessionCsrfTokenRepository"/>
   </constructor-arg>
   <property name="accessDeniedHandler" ref="ApplicationAccessDeniedHandler"/>
</bean>
```

The ToggleFilterWrapper class is responsible for wrapping the CsrfFilter and allows it to be disabled e.g. for development tests.

```
public class ToggleFilterWrapper implements Filter {
```

```
/** Logger instance. */
private static final Logger LOG = LoggerFactory.getLogger(ToggleFilterWrapper.class);
```

```
/**
 * The delegated Filter.
 */
private Filter delegateFilter;
```

```
/**
 * Is set if this filter is enabled.
 */
private Boolean enabled = Boolean.FALSE;
```

```
@Override
public void init(FilterConfig filterConfig) throws ServletException {
}
```

```
@PostConstruct
public void initialize() {
  if (!this.enabled) {
    String message =
        "****** FILTER " + this.delegateFilter
            + " HAS BEEN DISABLED! THIS FEATURE SHOULD ONLY BE USED IN DEVELOPMENT MODE ******";
    LOG.warn(message);
    System.err.println(message);
  }
}
```

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
 IOException,
ServletException {
  if (this.enabled) {
    this.delegateFilter.doFilter(request, response, chain);
  } else {
    chain.doFilter(request, response);
  }
}
```

```
@Override
public void destroy() {
}
```

```
/**
 * @param delegateFilter the filter to delegate to
 */
public void setDelegateFilter(Filter delegateFilter) {
  this.delegateFilter = delegateFilter;
}
```

```
/**
 * @param enabled the enabled flag
 */
public void setEnabled(Boolean enabled) {
  this.enabled = enabled;
}
```

```
  /**
   * @return disabled
   */
  public Boolean isEnabled() {
    return this.enabled;
  }
}
```

Server side - implementation

Retrieving the token on the server can be implemented as follows:

```
/**
 * The security REST service provides access to the csrf token, the authenticated user's meta-data.
 Furthermore, it
 * provides functionality to check permissions and roles of the authenticated user.
 *
 */
@Named("SecurityRestService")
@Transactional
public class SecurityRestServiceImpl {
```

```
/** Logger instance. */
private static final Logger LOG = LoggerFactory.getLogger(SecurityRestServiceImpl.class);
```

```
/**
```

```
 * Use {@link CsrfTokenRepository} for CSRF protection.
 */
private CsrfTokenRepository csrfTokenRepository;
```

```
/**
 * Retrieves the CSRF token from the server session.
 *
 * @param request {@link HttpServletRequest} to retrieve the current session from
 * @param response {@link HttpServletResponse} to send additional information
 * @return the Spring Security {@link CsrfToken}
 */
@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/csrftoken/")
@PermitAll
public CsrfToken getCsrfToken(@Context HttpServletRequest request, @Context HttpServletResponse
 response) {
  CsrfToken token = this.csrfTokenRepository.loadToken(request);
  if (token == null) {
    LOG.warn("No CsrfToken could be found - instanciating a new Token");
    token = this.csrfTokenRepository.generateToken(request);
    this.csrfTokenRepository.saveToken(token, request, response);
  }
  return token;
}
```

```
/**
 * Gets the profile of the user being currently logged in.
 *
 * @param request provided by the RS-Context
 * @return the {@link UserData} taken from the Spring Security context
 */
@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/currentuser/")
@PermitAll
public UserDetailsClientTo getCurrentUser(@Context HttpServletRequest request) {
  if (request.getRemoteUser() == null) {
    throw new NoActiveUserException();
  }
  return UserData.get().toClientTo();
}
```

```
  /**
   * @param csrfTokenRepository the csrfTokenRepository to set
   */
  @Inject
  public void setCsrfTokenRepository(CsrfTokenRepository csrfTokenRepository) {
    this.csrfTokenRepository = csrfTokenRepository;
  }
}
```

Client side - implementation

The client adds the CSRF token to all requests in oasp-security-service.service.js:

```
enableCsrfProtection = function () {
   return getSecurityRestService().getCsrfToken()
       .then(function (response) {
          var csrfProtection = response.data;
          // from now on a CSRF token will be added to all HTTP requests
          $http.defaults.headers.common[csrfProtection.headerName] = csrfProtection.token;
          currentCsrfProtection.set(csrfProtection.headerName, csrfProtection.token);
          return csrfProtection;
        }, function () {
          return $q.reject('Requesting a CSRF token failed');
        });
  };
```

4.8 S4-8

The application correctly enforces context-sensitive authorisation so as to not allow unauthorised manipulation by means of parameter tampering.

Purpose: Context-sensitive authorisation can take multiple parameters into account like e.g. the location of the user making the request to the system (if the location is not the office network, deny access). This is all nice, as long as a simple parameter tampering cannot bypass this control.

Solution: The OASP platform currently does not support context-sensitive authorisation.

5. S5 Basic Input Validation

This package describes basic input validation controls.

5.1 S5-1

The runtime environment is not susceptible to buffer overflows, or that security controls prevent buffer overflows.

Purpose: Memory overflow attacks (be it stack or heap overflows) allow the attacker to have read or read/write access to the applications memory, which can impact the integrity or accessibility of the application (see OWASP).

Solution: Secure by design. Java virtual machines protect its memory from direct user access and prevent this way buffer overflows as result of developers coding mistakes. Buffer overflows in the virtual machines itself are not improbable, but the machines itself are rigorously tested and the probability of such overflows to occur is negligible.

5.2 S5-2

All input validation failures result in input rejection and are logged.

Purpose: Data validation routines are the first line of defence against common attacks like e.g. injection.

Solution: This requirement should be understood as a good coding practice. Every time untrusted input data is validated, validation failures must always lead to either stopping the program flow itself (by throwing an exception with roll back of the uncommitted data) or continuing the program flow but rejecting the input (should ideally happen at the same place in code as the validation routine).

Another good coding practice is preferring whitelisting over blacklisting for the means of data validation. Whitelisting means allowing a program to continue if and only if the white listing (strongly narrowing) rules are met. In case of a mistake in the whitelisting function the program flow will terminate and the application will fail safe.

Blacklisting on the other hand terminates the program flow in case the black listing function recognizes the input as invalid or malicious. In case of a mistake in the blacklisting function, the program will continue with invalid (potentially malicious) input which can result in a security incident.

5.3 S5-3

All input validation or encoding routines are performed and enforced on the server side.

Purpose: Attacker can easily bypass all client side security controls.

Solution: This rule seems obvious at the first glance, but in fact it requires a certain level of development discipline.

It's a common development scenario, that user input is validated on the client side by means of Java Script (for performance reasons) or that multiple requests to the application (with different input data structures) are handled by one and the same server command. In both cases a strong server side validation should check and enforce at least the same rules as the client side does.

The challenge here is, that the server side validation routines can not be tested as part of functional tests running on the clients UI, as such tests will only touch the client side and its validation. The developers

should choose either unit tests or dynamic code analysis (manual test supported by proxy tools like Burp or OWASP ZAP) to ensure the server side validation works correct.

5.4 S5-4

The runtime environment is not susceptible to SQL Injection, or that security controls prevent SQL Injection.

Purpose: SQL injection problems arise from insufficient separation of the SQL query template from the template parametrization and happen in cases where the SQL statement is glued together using e.g. string concatenation of the SQL query and unvalidated user data. To get a good understanding about what SQL injection problems are please refer to the OWASP pages.

Solution: The hibernates JPA API offers a clean way to separate the SQL code from its parametrization which can be used to handle static queries. This approach has been already accurately described in the OASP4J documentation.

For dynamic queries it is advised to use QueryDSL (full SQL abstraction layer) as also described in the OASP4J documentation.

As often the case in security relevant topics, no one can say with 100% confidence that both approaches will always offer SQL injection protection. For practical purposes one can safely assume, that the probability of mistakenly writing SQL injection vulnerable code using one of the mentioned approaches is negligible.

5.5 S5-5

The runtime environment is not susceptible to LDAP Injection, or that security controls prevent LDAP Injection.

Purpose: Lightweight Directory Access Protocol (LDAP) Injection is an attack used to exploit web-based applications that construct LDAP statements based on user input. When an application fails to properly sanitize user input, it's possible to modify LDAP statements using a local proxy. This could result in the execution of arbitrary commands, such as granting permissions to unauthorized queries, and content modification inside the LDAP tree. The same advanced exploitation techniques available in SQL Injection can be similarly applied in LDAP Injection.

Solution: For Java consider using LDAP Spring. Using its filters and query builders considerably reduces risk, because dangerous characters are properly escaped. Do not write own LDAP authentication - use Spring Security instead.

5.6 S5-6

The runtime environment is not susceptible to OS Command Injection, or that security controls prevent OS Command Injection.

Purpose: You wouldn't like an attacker to execute arbitrary shell commands on your server, would you? OS command injection vulnerabilities in Java can arise by passing unvalidated user input to parametrize system calls using the class Runtime.

Solution: Our applications shouldn't usually have any need to to run OS commands at all. In rare cases such functionality is needed no parametrization with untrusted values should be allowed or the untrusted value should be validated by using a whitelisting (see S5-2) approach.

5.7 S5-7

The runtime environment is not susceptible to XML External Entity attacks or that security controls prevents XML External Entity attacks.

To understand the threat of external entity attacks please refer to OWASP pages. Also note, that in Java the XML streaming API enables external entities by default, so all Java applications that handle user XML input are potentially vulnerable to this kind of attack.

To turn external entities off use following approach:

5.7.1 dom4j

```
SAXReader reader = new SAXReader();
reader.setFeature("http://xml.org/sax/features/external-general-entities", false);
Document document = reader.read("<path to file>");
```

5.7.2 JDom

```
SAXBuilder builder = new SAXBuilder();
File xmlFile = new File("<path to file>");
builder.setExpandEntities(false);
Document document = builder.build(xmlFile);
```

5.7.3 JAXB

```
JAXBContext jc = JAXBContext.newInstance(UnmarshalledClass.class);
```

```
XMLInputFactory xif = XMLInputFactory.newFactory();
xif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
xif.setProperty(XMLInputFactory.SUPPORT_DTD, false); // if not needed
XMLStreamReader xsr = xif.createXMLStreamReader(new StreamSource("<path to file>"));
```

```
Unmarshaller unmarshaller = jc.createUnmarshaller();
UnmarshalledClass object = (UnmarshalledClass) unmarshaller.unmarshal(xsr);
```

5.8 S5-8

The runtime environment is not susceptible to XML Injections or that security controls prevents XML Injections.

Purpose: To get a good understanding of XML injections problems please refer to the OWASP pages.

Solution: The usual case how XML injection vulnerabilities are introduced in an application is manual string concatenation parametrized by unvalidated user input. The best solution to prevent such problems depends on the context, but is usually solved by tools and libraries offering full XML abstraction like JAXB, XStream (XML-Java marshalling and demarshalling), dom4j, JDOM (XML building and reading).

5.9 S5-9

All string variables placed into HTML or other web client code is either properly contextually encoded manually, or utilize templates that automatically encode contextually to ensure the application is not susceptible to reflected, stored and DOM Cross-Site Scripting (XSS) attacks.

Purpose: Please refer to the OWASP pages and the OWASP Top 10 list.

Solution: TODO: we definitely should write about how Angular handles this case and what can go wrong with using JQuery the wrong way. This is definitely a task for someone who knows both.

5.10 S5-10

The application is not susceptible to Remote File Inclusion (RFI) or Local File Inclusion (LFI) when content is used that is a path to a file.

Purpose: Please refer to OWASP pages to learn about Local and Remote File Inclusions.

Solution: This issue must be properly handled by the developer. A good hint how to handle this kind of situation is to canonicalize path names before validation.

6. S6 Basic Cryptography

chapter handles the solutions for secure cryptography at rest (so everything related to cryptography that is not about TLS and transport protection).

6.1 S6-1

All cryptographic modules fail securely, and errors are handled in a way that does not enable oracle padding.

Purpose: The opposite of fail secure is fail open meaning, that the attacker gains something despite the failure of some cryptographic functionality. A famous example of big security flaw is the padding oracle attack on symmetric ciphers working in the CBC mode. The only reason this attack was possible, was that the data decryption mechanism threw two different exceptions when the padding was incorrect and when the MAC was incorrect. This was a very small coding issue that led to the failure of the whole cryptosystem.

Solution: Honestly, if you deal with crypto on a level, that requires you to think about oracle padding attacks and similar, then you definitely are doing something wrong. If you need to encrypt/decrypt data on the code level, use the Spring Security functionality to do so, which is a higher abstraction layer than the JCA/JCE libraries and is less prone to coding errors.

6.2 S6-2

Verify that cryptographic algorithms used by the application have been validated against FIPS 140-2 or an equivalent standard.

Purpose: FIPS 140-2 is an American standard for cryptography modules. Using only FIPS approved libraries raises the trust that the crypto is working correctly.

Solution: FIPS compliant libraries for Java are rather expensive tools you can buy from big market players with one exception - Bouncy Castle - which is free, certified and available for all.

7. S7 Basic Error Handling

This package describes basic error handling controls.

7.1 S7-1

The application does not output error messages or stack traces containing sensitive data that could assist an attacker, including session id, software/framework versions and personal information.

Purpose: Developers usually like to see ad hoc why the system failed (when it failed). This usually leads to stack trace information being directly exposed in response body. This information can assist the attacker in multiple ways.

Solution: Secure by default. The OASP platform introduces the class `io.oasp.module.rest.service.impl.RestServiceExceptionFacade` (preloaded by the JAX-RS `@Provider` annotation) that handles how exceptions are visible to the client application. The secure default is, that no stack traces are exposed to the client. It can be eventually overwritten for development purposes by setting a the flag `exposeInternalErrorDetails` on `true`, but must never be used this way on the production environment.

```
if (this.exposeInternalErrorDetails) {
      message = getExposedErrorDetails(error);
} else {
      message = error.getLocalizedMessage();
}
```

7.2 S7-2

Time sources should be synchronized to ensure logs have the correct time.

Purpose: This is crucial for every system integrity analysis, if we have to extract information from multiple logs on different machines (very true for micro service architectures).

Solution: The solution for this problem lies far below the OASP platform, in the usage of the NTP protocol.

Note: You need to handle the time synchronization topic even if you use infrastructure of some cloud service providers (e.g. AWS).

8. S8 Basic Data Protection

This package describes basic data protection controls.

8.1 S8-1

All forms containing sensitive information have disabled client side caching, including autocomplete features.

Purpose: All sensitive data cannot be cached beyond the scope of the users request, as this would open door for a potential data leak.

Solution: First of all the system owner must define what information is sensitive in the systems context. Please note: there is no clear definition of what sensitive information is. Usually this category includes (this list is not exhaustive):

- law protected data,
- data protected by a customers policy,
- passwords.

For all form fields containing sensitive information the browser caching must be disabled with `autocomplete=off`.

> **Note**
>
> The Chrome browser ignores the `autocomplete=off` attribute. If it is explicitly required by the system owner to suppress the client side caching in Chrome as well, this web page describes a workaround.

8.2 S8-2

All sensitive data is sent to the server in the HTTP message body (i.e., URL parameters are never used to send sensitive data).

Purpose: There might be a technical need to turn on request logging on the web server on any part of the application platform. Should this be the case, we will not want any sensitive data to be exposed in the application logs.

Solution: For an explanation what sensitive data is please refer to solution S8-1. Compliance with this requirement is not be enforced by the OASP platform. The systems architects and developers must follow the rule of never using URL parameters to send sensitive data.

8.3 S8-3

The application sets appropriate anti-caching headers as per the risk of the application, such as the following: `Expires: Tue, 03 Jul 2001 06:00:00 GMT ; Last-Modified: {now} GMT ; Cache-Control: no-store, no-cache, must-revalidate, max-age=0 ; Cache-Control: post-check=0, pre-check=0 ; Pragma: no-cache`

Purpose: It is usually a good idea to turn off caching for every request to dynamic system data (like HTML and REST calls) per default. It is not allowed to store or cache sensitive data in the users browser,

as they could be recovered by the attacker with physical access to the user agent (after the legitimate user session ends).

> **Note**
>
> You can (and probably should) use caching for static, non-sensitive data like images, JS and CSS files, …

Solution: Spring Security sets some default response headers which is a nice secure default of the Spring platform itself. Header definition like `Cache-Control: no-cache, no-store, max-age=0, must-revalidate` and `Pragma: no-cache` is enough to turn off caching for the modern browsers. All the developer needs to do is not disable the default headers definition.

8.4 S8-4

Data stored in client side storage (such as HTML5 local storage, session storage, IndexedDB, regular cookies or Flash cookies) does not contain sensitive data or PII.

Purpose: As in S8-3, this could expose sensitive data to an attacker with physical access to the user agent.

Solution: This needs to be handled by the DEV team this way.

9. S9 Basic Communication Security

This package describes basic communication security controls.

9.1 S9-1

A path can be built from a trusted CA to each Transport Layer Security (TLS) server certificate, and that each server certificate is valid.

Purpose: Valid certificates on the path mitigate the threat of Man in the Middle (MitM) attacks.

Solution: Buy certificates from the official RA (Registration Authority) or go for Let's Encrypt if possible. Deploy the certificates properly. If the work has been done properly can be manually checked via SSL Labs (host available in internet) or Test SSL (host not available in internet).

9.2 S9-2

TLS is used for all connections (including both external and backend connections) that are authenticated or that involve sensitive data or functions, and does not fall back to insecure or unencrypted protocols. Ensure the strongest alternative is the preferred algorithm.

Purpose: TLS offers message confidentiality, integrity and sever authentication. If you don't use TLS, you don't have any of them.

Solution: Use TLS. Everywhere.

9.3 S9-3

HTTP Strict Transport Security headers are included on all requests and for all subdomains, such as `Strict-Transport-Security: max-age=15724800; includeSubdomains`.

Purpose: Even if the application uses TLS, it is a common practice to deploy the application on both port 80 (HTTP, not TLS) and 443 (HTTPS). This can lead to multiple problems like:

- To access the application the user inputs an URL like `myapplication.com`. The user agent fires a HTTP request on port 80 with URL `http://myapplication.com`. The server will most probably redirect it with response 301 to `https://myapplication.com` (clear mistake if it does not). Still, the redirect leaves a hole for the attacker to catch the request and redirect it elsewhere due to in inexistent server authentication.
- Due to a DEV mistake the application loads JS files from the server over HTTP, even though the REST requests are TLS protected introducing mixed content problems.

Solution: HTTP Strict Transport Security (HSTS) header will register the given domain as a STS domain on the user agent (browser) for the defined period of time (15724800 seconds = 182 days). The browser will then make an internal redirect (307) for every HTTP call going to this domain or any subdomain (if `includeSubdomains` turned on).

> **Note**
>
> This actually means, that you can't go back to HTTP any more, as the user agent will not allow HTTP calls any more.

Spring Security offers a quick and easy way to turn HSTS on.

9.4 S9-4

Forward secrecy ciphers are in use to mitigate passive attackers recording traffic.

Purpose: A passive attacker might choose to record all TLS encrypted traffic waiting for an opportunity to steal the private key from the server. This would normally expose all past communication between the client and the server.

TLS 1.2 introduces the idea of (perfect) forward secrecy ciphers, which prevent previous communication from being exposed if the certificate leaks. The ciphers use the Diffie Hellman key exchange algorithm to negotiate the encryption key between two parties every time a new TLS session is started. This key remains a secret even if the attacker comes into the possession of the servers private key.

Solution: Make sure you have TLS 1.2 turned on. Make sure, you have support for DHE or ECDHE ciphers in your configuration. Test your configuration with SSL Labs (host available in internet) or Test SSL (host not available in internet) to make sure this is really the case.

9.5 S9-5

Proper certification revocation, such as Online Certificate Status Protocol (OCSP) Stapling, is enabled and configured.

Purpose: How does the client make sure, that the servers certificate is still valid? It must contact the CA and ask, if the certificate has been revoked or not. This reveals to the CA, that the client wants to access a specific server. This is an information leak. OCSP stapling is meant to prevent it

Solution: Turn OCSP stapling on your server.

9.6 S9-6

Only strong algorithms, ciphers, and protocols are used, through all the certificate hierarchy, including root and intermediary certificates of your selected certifying authority.

Purpose: Chain is only as strong as the weakest link.

Solution: Test your configuration with SSL Labs (host available in internet) or Test SSL (host not available in internet) to make sure the configuration is ok. Make configuration changes if necessary.

9.7 S9-7

TLS settings are in line with current leading practice, particularly as common configurations, ciphers, and algorithms become insecure.

Purpose: Chain is only as strong as the weakest link.

Solution: Test your configuration with SSL Labs (host available in internet) or Test SSL (host not available in internet) to make sure the configuration is ok. Make configuration changes if necessary.

10. S10 Basic Security Configuration

This package describes basic HTTP security controls.

10.1 S10-1

The application accepts only a defined set of required HTTP request methods, such as GET and POST are accepted, and unused methods (e.g. TRACE, PUT, and DELETE) are explicitly blocked.

Purpose: Exposing more then needed just raises the attack surface.

Solution: In Apache Tomcat, security is enforced by way of security constraints that are built into the Java Servlet specification. These are to be contained within the web.xml configuration file. Following definition works with Tomcat 7+ servers and enables only HEAD, GET, POST and PUT methods to be called.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>restricted methods</web-resource-name>
        <url-pattern>/*</url-pattern>
        <http-method-omission>GET</http-method-omission>
        <http-method-omission>HEAD</http-method-omission>
        <http-method-omission>PUT</http-method-omission>
        <http-method-omission>POST</http-method-omission>
    </web-resource-collection>
    <auth-constraint />
</security-constraint>
```

10.2 S10-2

Every HTTP response contains a content type header specifying a safe character set (e.g., UTF-8).

Purpose: If the response does not define the character set, the browser will try to guess it. This might open door for the attacker to try e.g. UTF-7 encoding to bypass sanitization and XSS prevention functionality.

Solution: The fastest way to satisfy the requirement is to set following filter definition in the `web.xml` file:

```
<filter>
    <filter-name>SetCharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <async-supported>true</async-supported>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>SetCharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

10.3 S10-3

The HTTP headers or any part of the HTTP response do not expose detailed version information of system components.

Purpose: Knowledge of the servers version unnecessarily supports the attacker. Knowing the version he can go to e.g. Exploit DB and search for known vulnerabilities and exploits.

Solution: Make sure the servers version never appear in server responses. For Tomcat this can be done by putting following into the `server.xml` file, to the host section:

```
<Valve className="org.apache.catalina.valves.ErrorReportValve"
showReport="false"
showServerInfo="false"/>
```

10.4 S10-4

All API responses contain `X-Content-Type-Options: nosniff` and `Content-Disposition: attachment; filename="api.json"` (or other appropriate filename for the content type).

Purpose: Did you know that your browser might ignore the content type for the response provided by the server? It can. You would not like to download a text file and have the browser recognize it as JS (and execute), would you? This is the reason for using the X-Content-Type-Options header.

With the Content-Disposition header you tell the browser to treat the downloaded file as an attachment (show the Save As dialogue) and not inline the content (show it in the browser, execute).

Solution: `X-Content-Type-Options: nosniff` is a secure default of Spring Security.

Content-Disposition header needs to be set by the dev manually when needed. This can be done by using `setHeader` method of the `HttpServletResponse` object like this:

```
response.setHeader("Content-Disposition", "attachment; filename=Downloaded_file.txt");
```

10.5 S10-5

Content security policy (CSPv2) is in place that helps mitigate common DOM, XSS, JSON, and JavaScript injection vulnerabilities.

Purpose: CSP is powerful way to implement another level of protection from common client side attacks (XSS, clickjacking, …).

Solution: Unfortunately is not easy to implement and different systems will require different CSP header definitions. Use the generator to create a header definition to start with. Turn it on in report only mode first. Create a reporting point. Then incrementally make the header more and more strict watching the reports provided by the user agents.

TODO: make an example for this based on My Thai Star app.

10.6 S10-6

The `X-XSS-Protection: 1; mode=block` header is in place.

Purpose: The X-XSS-Protection header can stop some forms of reflected Cross-Site Scripting attacks in IE and Chrome-based browsers. In fact this functionality is turned on by default. What the header

definition above does additionally is turning the block mode on. The difference is, that in default configuration the browser will try to sanitize the attack vector (which, as the history has shown, can lead to XSS attacks, that would not be possible otherwise). In the blocking mode the browser will **stop the rendering of the page** if an attack is discovered.

Solution: This is a secure default of Spring Security.

11. S11 Basic Files and Ressources Protection

This package describes file and resources protection controls.

11.1 S11-1

URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.

Purpose: To understand the threat of unvalidated redirects and forwards please refer to the OWASP pages.

Solution: There shouldn't be usually any need to perform redirects or forwards to sites or pages parametrized by the user input. In the unlikely case the functionality is needed, use a whitelisting approach to validate the user input.

11.2 S11-2

Untrusted file data submitted to the application is not used directly with file I/O commands, particularly to protect against path traversal, local file include, file mime type, and OS command injection vulnerabilities.

Purpose: To understand the threat please refer to following pages:

- path traversal
- local file inclusion
- OS command injection

Solution: This is not something the OASP platform can help with. Solution to this problem lies in the hands of system developers, who must use strong whitelisting/sanitization approaches when dealing with file paths, file names and content of untrusted data.

11.3 S11-3

Files obtained from untrusted sources are scanned by antivirus scanners to prevent upload of known malicious content.

Purpose: Sometimes the attacks are so primitive, that even the antivirus can help…

Solution: Malware scanner integration details depend on multiple things like:

- vendor of the scanner,
- what the scanner scans (HTTP stream, files),
- is the scanner accessible locally or is it an online service.

Description of the full scope of integration details is outside the scope of this guide.

11.4 S11-4

Untrusted data is not used within inclusion, class loader, or reflection capabilities to prevent remote/local file inclusion vulnerabilities.

Purpose: Please refer to the OWASP pages regarding local and remote file inclusion attacks.

Solution: Treat this as an requirement to be included in your software development lifecycle, as the proper implementation lies in the hands of the developer.

11.5 S11-5

Untrusted data is not used within cross-domain resource sharing (CORS) to protect against arbitrary remote content.

Purpose: Arbitrary remote content might impact the integrity of the system.

Solution: Treat this as an requirement to be included in your software development lifecycle, as the proper implementation lies in the hands of the developer.

11.6 S11-6

The application code does not execute uploaded data obtained from untrusted sources.

Purpose: Execution of uploaded data might lead to remote code execution.

Solution: Treat this as an requirement to be included in your software development lifecycle, as the proper implementation lies in the hands of the developer.

11.7 S11-7

Do not use Flash, Active-X, Silverlight, NACL, client-side Java or other client side technologies not supported natively via W3C browser standards.

Purpose: For example - you should not use Flash because of more than 1000 reasons.

Solution: The OASP platform does not rely on any non-W3C technologies and the dev team should not use them either.

12. S12 Basic Web Services Security

This chapter contains basic security controls around web service protection.

12.1 S12-1

The same encoding style is used between the client and the server.

Purpose: Different encodings between client and server can have security implications and have been used in the past to bypass validation and WAFs to perform XSS attacks.

Solution: Spring and OASP use UTF-8 on both client and server side per default.

> **Note**
>
> JSON request with `Content-Type` set to `application/json` must not explicitly set the encoding type to UTF-8.

12.2 S12-2

Access to administration and management functions within the Web Service Application is limited to web service administrators.

Purpose: Unauthorized users accessing administration or management functions of the system can disturb the systems integrity.

Solution: This requirement must be tested and verified by the dev team.

12.3 S12-3

XML or JSON schema is in place and verified before accepting input.

Purpose: Schema validation is the first level of defence against attacks on the application logic.

Solution: OASP platform encourages the developers to use XML schema validation wherever possible. The problem with JSON schema is: there is no official standard yet, only implementations that base on IETF Draft proposals. Use it based on your own risk analysis process.

12.4 S12-4

All input is limited to an appropriate size limit.

Purpose: Some e.g. DOS attacks involve sending huge payloads to the server.

Solution: Spring Boot limits the size of the request and uploaded files per default over properties:

```
max-file-size specifies the maximum size permitted for uploaded files. The default is 1MB.
max-request-size specifies the maximum size allowed for multipart/form-data requests. The default is
 10MB.
```

12.5 S12-5

SOAP based web services are compliant with Web Services-Interoperability (WS-I) Basic Profile at minimum.

Purpose: Web services, which do not use a library claiming compliance with the WS-I Basic Profile at minimum, are more likely to behave in an unforeseen way, which can lead to security issues.

Solution: Apache CXF, used by the OASP platform, claims compliance with the WS-I profile.

13. S13 Basic Configuration

This package describes configuration controls to ensure the software is up to date, hardened and secure by default if possible.

13.1 S13-1

System does not use libraries or frameworks with known vulnerabilities.

Purpose: To understand the threat of using components with known vulnerabilities please refer to the OWASP pages. Here you can see how the internet articles will write about developers failure of updating critical framework libraries.

Solution: Ensuring the system does not use libraries or frameworks with known vulnerabilities is not a one-time activity. Such security checks must run regularly. OWASP Dependency Check is the most popular open source tool that can help with that in terms of Java libraries. Retire.js can deal with JavaScript.

Consider using these tools as part of your CI/CD pipeline e.g. over Jenkins plugins.

Part I. OWASP ASVS V3.0.1

Direct link to the standard

The **OWASP Application Security Verification Standard (ASVS)** Project provides a basis for testing web application technical security controls and also provides developers with a list of requirements for secure development.

The primary aim of the OWASP ASVS Project is to normalize the range in the coverage and level of rigor available in the market when it comes to performing Web application security verification using a commercially-workable open standard. The standard provides a basis for testing application technical security controls, as well as any technical security controls in the environment, that are relied on to protect against vulnerabilities such as Cross-Site Scripting (XSS) and SQL injection. This standard can be used to establish a level of confidence in the security of Web applications. The requirements were developed with the following objectives in mind:

- **Use as a metric** - Provide application developers and application owners with a yardstick with which to assess the degree of trust that can be placed in their Web applications,
- **Use as guidance** - Provide guidance to security control developers as to what to build into security controls in order to satisfy application security requirements, and
- **Use during procurement** - Provide a basis for specifying application security verification requirements in contracts.

The ASVS standard defines following assurance levels of the applications security:

- **Level 0 (cursory)** - optional certification indicating, that the application has passed some type of verification. The standard does not define the scope of this level. The project owner can define his own minimum criteria (often as subset of the Level 1 requirements).
- **Level 1 (opportunistic)** - application defends against vulnerabilities, that are easy to discover.
- **Level 2 (standard)** - application defends against vulnerabilities whose existence poses moderate to serious risk.
- **Level 3 (advanced)** - application defends against all advanced vulnerabilities and demonstrates principles of good security design.

NOTE:

- To achieve Level 1 certification the application must pass **ALL** Level 1 requirements.
- To achieve Level 2 certification the application must pass **ALL** Level 1 and Level 2 requirements.

14. Level 1 Requirements

14.1 V1 Architecture

Design and threat modelling Solution S1.

- **V1.1**: Verify that all application components are identified and are known to be needed. (→ S1-1)

14.2 V2 Authentication

See Solution S2.

- **V2.1**: Verify all pages and resources require authentication except those specifically intended to be public (Principle of complete mediation). (→ S2-1)
- **V2.2**: Verify that forms containing credentials are not filled in by the application. Pre-filling by the application implies that credentials are stored in plaintext or a reversible format, which is explicitly prohibited. (→ S2-2)
- **V2.4**: Verify all authentication controls are enforced on the server side. (→ S2-3)
- **V2.6**: Verify all authentication controls fail securely to ensure attackers cannot log in. (→ S2-4)
- **V2.7**: Verify password entry fields allow, or encourage, the use of passphrases, and do not prevent password managers, long passphrases or highly complex passwords being entered. (→ S2-9)
- **V2.8**: Verify all account identity authentication functions (such as update profile, forgot password, disabled / lost token, help desk or IVR) that might regain access to the account are at least as resistant to attack as the primary authentication mechanism. (→ S2-10)
- **V2.9**: Verify that the changing password functionality includes the old password, the new password, and a password confirmation. (→ S2-11)
- **V2.16**: Verify that credentials are transported using a suitable encrypted link and that all pages/ functions that require a user to enter credentials are done so using an encrypted link. (→ S2-5)
- **V2.17**: Verify that the forgotten password function and other recovery paths do not reveal the current password and that the new password is not sent in clear text to the user. (→ S2-6)
- **V2.18**: Verify that information enumeration is not possible via login, password reset, or forgot account functionality. (→ S2-7)
- **V2.19**: Verify there are no default passwords in use for the application framework or any components used by the application (such as “admin/password”). (→ S2-8)
- **V2.20**: Verify that anti-automation is in place to prevent breached credential testing, brute forcing, and account lockout attacks. (→ S2-12)
- **V2.22**: Verify that forgotten password and other recovery paths use a TOTP or other soft token, mobile push, or other offline recovery mechanism. Use of a random value in an e-mail or SMS should be a last resort and is known weak. (→ S2-13)
- **V2.24**: Verify that if shared knowledge based questions (also known as "secret questions") are required, the questions do not violate privacy laws and are sufficiently strong to protect accounts from malicious recovery. (→ S2-14)

- **V2.27**: Verify that measures are in place to block the use of commonly chosen passwords and weak passphrases. (→ S2-15)

14.3 V3 Session management

Handled by Solution S3.

- **V3.1** Verify that there is no custom session manager, or that the custom session manager is resistant against all common session management attacks. (→ S3-1)
- **V3.2** Verify that sessions are invalidated when the user logs out. (→ S3-2)
- **V3.3** Verify that sessions timeout after a specified period of inactivity. (→ S3-3)
- **V3.5** Verify that all pages that require authentication to access them have logout links. (→ S3-4)
- **V3.6** Verify that the session id is never disclosed in URLs, error messages, or logs. This includes verifying that the application does not support URL rewriting of session cookies. (→ S3-5)
- **V3.7** Verify that all successful authentication and re-authentication generates a new session and session id. (→ S3-6)
- **V3.12** Verify that session ids stored in cookies have their path set to an appropriately restrictive value for the application, and authentication session tokens additionally set the “HttpOnly” and “secure” attributes. (→ S3-7)
- **V3.16** Verify that the application limits the number of active concurrent sessions. (→ S3-8)
- **V3.17** Verify that an active session list is displayed in the account profile or similar of each user. The user should be able to terminate any active session. (→ S3-9)
- **V3.18** Verify the user is prompted with the option to terminate all other active sessions after a successful change password process. (→ S3-10)

14.4 V4 Access control

Handled by Solution S4.

- **V4.1** Verify that the principle of least privilege exists - users should only be able to access functions, data files, URLs, controllers, services, and other resources, for which they possess specific authorization. This implies protection against spoofing and elevation of privilege. (→ S4-1, S4-2)
- **V4.4** Verify that access to sensitive records is protected, such that only authorized objects or data is accessible to each user (for example, protect against users tampering with a parameter to see or alter another user’s account). (→ S4-3)
- **V4.5** Verify that directory browsing is disabled unless deliberately desired. Additionally, applications should not allow discovery or disclosure of file or directory metadata, such as Thumbs.db, .DS_Store, .git or .svn folders. (→ S4-4)
- **V4.8** Verify that access controls fail securely. (→ S4-5)
- **V4.9** Verify that the same access control rules implied by the presentation layer are enforced on the server side. (→ S4-6)

- **V4.13** Verify that the application or framework uses strong random anti-CSRF tokens or has another transaction protection mechanism. (→ S4-7)
- **V4.16** Verify that the application correctly enforces context-sensitive authorisation so as to not allow unauthorised manipulation by means of parameter tampering. (→ S4-8)

14.5 V5 Input validation

Handled by Solution S5

- **V5.1** Verify that the runtime environment is not susceptible to buffer overflows, or that security controls prevent buffer overflows. (→ S5-1)
- **V5.3** Verify that server side input validation failures result in request rejection and are logged. (→ S5-2)
- **V5.5** Verify that input validation routines are enforced on the server side. (→ S5-3)
- **V5.10** Verify that all SQL queries, HQL, OSQL, NOSQL and stored procedures, calling of stored procedures are protected by the use of prepared statements or query parametrization, and thus not susceptible to SQL injection. (→ S5-4)
- **V5.11** Verify that the application is not susceptible to LDAP Injection, or that security controls prevent LDAP Injection. (→ S5-5)
- **V5.12** Verify that the application is not susceptible to OS Command Injection, or that security controls prevent OS Command Injection. (→ S5-6)
- **V5.13** Verify that the application is not susceptible to Remote File Inclusion (RFI) or Local File Inclusion (LFI) when content is used that is a path to a file. (→ S5-10)
- **V5.14** Verify that the application is not susceptible to common XML attacks, such as XPath query tampering, XML External Entity attacks, and XML injection attacks. (→ S5-7, S5-8)
- **V5.15** Ensure that all string variables placed into HTML or other web client code is either properly contextually encoded manually, or utilize templates that automatically encode contextually to ensure the application is not susceptible to reflected, stored and DOM Cross-Site Scripting (XSS) attacks. (→ S5-9)
- **V5.22** Make sure untrusted HTML from WYSIWYG editors or similar are properly sanitized with an HTML sanitizer and handle it appropriately according to the input validation task and encoding task. (→ S5-9)

14.6 V7 Cryptography at rest verification requirements

Handled by Solution S6

- **V7.2** Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable oracle padding. (→ S6-1)
- **V7.7** Verify that cryptographic algorithms used by the application have been validated against FIPS 140-2 or an equivalent standard. (→ S6-2)

14.7 V8 Error handling and logging verification requirements

Handled by Solution S7

- **V8.1** Verify that the application does not output error messages or stack traces containing sensitive data that could assist an attacker, including session id, software/framework versions and personal information. (→ S7-1)
- **V8.13** Time sources should be synchronized to ensure logs have the correct time. (→ S7-2)

14.8 V9 Data protection verification requirements

Handled by Solution S8 * **V9.1** Verify that all forms containing sensitive information have disabled client side caching, including autocomplete features. (→ S8-1) * **V9.3** Verify that all sensitive data is sent to the server in the HTTP message body or headers (i.e., URL parameters are never used to send sensitive data). (→ S8-2) * **V9.4** Verify that the application sets appropriate anti-caching headers as per the risk of the application, such as the following: `Expires: Tue, 03 Jul 2001 06:00:00 GMT ; Last-Modified: {now} GMT ; Cache-Control: no-store, no-cache, must-revalidate, max-age=0 ; Cache-Control: post-check=0, pre-check=0 ; Pragma: no-cache.` (→ S8-3) * **V9.9** Verify that data stored in client side storage (such as HTML5 local storage, session storage, IndexedDB, regular cookies or Flash cookies) does not contain sensitive data or PII. (→ S8-4)

14.9 V10 Communications security verification requirements

Handled by Solution S9

- **V10.1** Verify that a path can be built from a trusted CA to each Transport Layer Security (TLS) server certificate, and that each server certificate is valid. (→ S9-1)
- **V10.3** Verify that TLS is used for all connections (including both external and backend connections) that are authenticated or that involve sensitive data or functions, and does not fall back to insecure or unencrypted protocols. Ensure the strongest alternative is the preferred algorithm. (→ S9-2)
- **V10.11** Verify that HTTP Strict Transport Security headers are included on all requests and for all subdomains, such as `Strict-Transport-Security: max-age=15724800; includeSubdomains`. (→ S9-3)
- **V10.13** Ensure forward secrecy ciphers are in use to mitigate passive attackers recording traffic. (→ S9-4)
- **V10.14** Verify that proper certification revocation, such as Online Certificate Status Protocol (OCSP) Stapling, is enabled and configured. (→ S9-5)
- **V10.15** Verify that only strong algorithms, ciphers, and protocols are used, through all the certificate hierarchy, including root and intermediary certificates of your selected certifying authority. (→ S9-6)
- **V10.16** Verify that the TLS settings are in line with current leading practice, particularly as common configurations, ciphers, and algorithms become insecure. (→ S9-7)

14.10 V11 HTTP security configuration verification requirements

Handled by Solution S10

- **V11.1** Verify that the application accepts only a defined set of required HTTP request methods, such as GET and POST are accepted, and unused methods (e.g. TRACE, PUT, and DELETE) are explicitly blocked. (→ S10-1)

- **V11.2** Verify that every HTTP response contains a content type header specifying a safe character set (e.g., UTF-8, ISO 8859-1). (→ S10-2)
- **V11.5** Verify that the HTTP headers or any part of the HTTP response do not expose detailed version information of system components. (→ S10-3)
- **V11.6** Verify that all API responses contain `X-Content-Type-Options: nosniff` and `Content-Disposition: attachment; filename="api.json"` (or other appropriate filename for the content type). (→ S10-4)
- **V11.7** Verify that a content security policy (CSPv2) is in place that helps mitigate common DOM, XSS, JSON, and JavaScript injection vulnerabilities. (→ S10-5)
- **V11.8** Verify that the `X-XSS-Protection: 1; mode=block` header is in place. (→ S10-6)

14.11 V16 Files and resources verification requirements

Handled by Solution S11

- **V16.1** Verify that URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content. (→ S11-1)
- **V16.2** Verify that untrusted file data submitted to the application is not used directly with file I/O commands, particularly to protect against path traversal, local file include, file mime type, and OS command injection vulnerabilities. (→ S11-2)
- **V16.3** Verify that files obtained from untrusted sources are validated to be of expected type and scanned by antivirus scanners to prevent upload of known malicious content. (→ S11-3)
- **V16.4** Verify that untrusted data is not used within inclusion, class loader, or reflection capabilities to prevent remote/local file inclusion vulnerabilities. (→ S11-4)
- **V16.5** Verify that untrusted data is not used within cross-domain resource sharing (CORS) to protect against arbitrary remote content. (→ S11-5)
- **V16.8** Verify the application code does not execute uploaded data obtained from untrusted sources. (→ S11-6)
- **V16.9** Do not use Flash, Active-X, Silverlight, NACL, client-side Java or other client side technologies not supported natively via W3C browser standards. (→ S11-7)

14.12 V18 Web services verification requirements

Partially handled by Solution S12

- **V18.1** Verify that the same encoding style is used between the client and the server. (→ S12-1)
- **V18.2** Verify that access to administration and management functions within the Web Service Application is limited to web service administrators. (→ S12-2)
- **V18.3** Verify that XML or JSON schema is in place and verified before accepting input. (→ S12-3)
- **V18.4** Verify that all input is limited to an appropriate size limit. (→ S12-4)

- **V18.5** Verify that SOAP based web services are compliant with Web Services-Interoperability (WS-I) Basic Profile at minimum. This essentially means TLS encryption. (→ S12-5)
- **V18.6** Verify the use of session-based authentication and authorization. Please refer to sections 2, 3 and 4 for further guidance. Avoid the use of static "API keys" and similar. (→ S2, S3, S4)
- **V18.7** Verify that the REST service is protected from Cross-Site Request Forgery via the use of at least one or more of the following: ORIGIN checks, double submit cookie pattern, CSRF nonces, and referrer checks. (→ S4-7)

14.13 V19 Configuration verification requirements

Handled by Solution S13

- **V19.1** All components should be up to date with proper security configuration(s) and version(s). This should include removal of unneeded configurations and folders such as sample applications, platform documentation, and default or example users. (→ S13-1)

Part II. OWASP Top 10 2017

Direct link to the Top 10 list

The Top 10 list is the most popular project of the OWASP community and is often used as the entry point into the application security domain. The list defines 10 biggest security risks the web applications are facing (de facto meaning - 10 most common security problems) and is updated every ~3 years.

OWASP Top 10 list is greatly misused world wide. The purpose of the list is to raise the awareness among developers and inspire the development culture. The Top 10 list is not a standard (that's what OWASP ASVS is for) and should not be used for security audit or security testing. The list is also insufficient to consider the compliant applications secure.

Still the OWASP Top 10 list compliance can be found among the most common security requirements and customer requests.

OWASP Top 10 2017 defines following risks:

- **A1-Injection** - Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization. **Solutions:** S5-4, S5-5, S5-6, S5-8.
- **A2-Broken Authentication** - Application functions related to authentication are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities. **Solutions:** S2.
- **A3-Sensitive Data Exposure** - Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser. **Solutions:** S4, S9, S7-1
- **A4-XML External Entities (XXE)** - Many older or poorly configured (or simply java-based) XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks. **Solutions:** S5-7
- **A5-Broken Access Control** - Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc. **Solutions:** S4
- **A6-Security Misconfiguration** - Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date. **Solutions:** This point goes far beyond the scope of this guide, as it addresses problems of the proper server configuration. The application will not be secure, if the servers it is running on are not properly configured (hardened). On of many resources available online that can help to properly configure the servers are the CIS Benchmarks.
- **A7-Cross-Site Scripting (XSS)** - XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute

scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites. **Solutions:** S5-9

- **A8-Insecure deserialization** - Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks. **Solutions:** TODO, as this is not yet covered by OWASP ASVS 3.0.1.
- **A9-Using Components with Known Vulnerabilities** - Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts. **Solutions:** S13-1
- **A10-Insufficient Logging & Monitoring** - Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring. **Solutions**: S5-2, S7, TODO - there should be more than this.