[MANDATORY SECURITY] GUIDELINES

GL–018 v.01

SECURE GUIDELINE J2EE

COVER

Title	Secure Guideline J2EE
Classification	Mandatory Security Guidelines
Document code	GL-018 v.01
Approved by	Nexi Group CISO
Approval date	12-06-2023
Date of entry into force	12-06-2023

UPDATES

Version	Date	Code	Updates
1	12-06-2023	GL-018 v.01	First issue

SUMMARY

1 INTRODUCTION

The purpose of this document is to describe the security requirements that should be implemented in order to develop secure web applications using J2EE and Spring.

These requirements originate from the use of standard and internationally recognized methodologies such as OWASP (The Open Web Application Security Project).

The following references were used during the writing of this document:

- “OWASP Development Guide” – OWASP Foundation
- “OWASP Testing Guide” – OWASP Foundation
- “OWASP Cheat Sheets Series” – OWASP Foundation
- “OWASP Secure Coding Practices” – OWASP Foundation

In particular, the security requirements to be implemented are based on the security tests described in the OWASP Testing Guide. Those will also be the reference for the subsequent security assessment phase of the software produced.

The set of requirements to be implemented were divided by analysis area (authentication, authorization, data validation, etc.), according to the OWASP Testing Guide v4.

Each control to be implemented includes a table that schematically summarizes the following information:

Requirement ID	Analysis Area 001
Priority	Priority of the requested requirement
Description	Description of the requirement to be implemented
Java/J2EE	Example of controls to implement using Java / J2EE
Java/Spring	Example of controls to implement using Spring specific solutions
Reference for verification	OWASP reference for verifying the requirement to be implemented

2 DEFINITIONS AND ABBREVIATIONS

Term	Description
XMLHttpRequest	JavaScript object that is used for AJAX asynchronous requests; allows to make HTTP GET or POST requests to the server application and receive the response without having to wait for the entire page to be refreshed.
Cross Site Request Forgery (CSRF o XSRF)	Attack through which the attacker is able to force the victim to perform actions on the vulnerable site where he is authenticated, without the victim being aware of it.
Cross Site Scripting (XSS)	An attack in which the attacker injects HTML / JavaScript code which is then executed in the victim's browser in the context of the vulnerable site.
UI Redressing	Fraudulent technique that tries to offer the victim a page that displays harmless elements that trick the user to perform actions with the mouse or to fill in forms. In reality, these actions are redirected to one or more transparent frames that contain the pages of a site that is the subject of the attack itself to which the victim user is logged in (eg Facebook Like or user entry page in an administration interface).
CORS	Cross Origin Resource Sharing. The way in which browsers allow the reading of data outside the Same Origin Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

3 REQUIREMENTS DESCRIPTION

Below is a brief description of the categories of the requirements:

3.1 AUTHENTICATION

In the field of security, authentication is the process designed to verify the digital identity of whoever is interacting with the application. The purpose of authentication controls on application users is to uniquely associate the users with their identity on the system, in order that they can access their data. This also means preventing access to resources by users who do not have access credentials.

Following are reported some general best practices in addition to the proposed requirements useful for guaranteeing the security of the application authentication mechanisms:

- Whenever possible, reuse standard authentication services, which have been tested for security.
- It is good practice to use a centralized implementation for all authentication controls, including libraries used to call external authentication services.
- All authentication checks must fail securely, returning the system to the state prior to the failed request. Ex. close all files opened during execution and prior to failure.
- All administrative and account management functions should not normally be accessible from the Internet in the same way a user is authenticated: make this functionality available only from within the corporate network or via VPN if it is necessary to expose them on the Internet.
- Connections to external systems should be authenticated (do not allow implicit trust to third party applications).
- The last login (successful or unsuccessful) of a user account must be reported to the user on their next successful login.
- If you are using third-party code for authentication, carefully inspect the code to make sure it is not affected by any malicious code.

3.2 ACCESS CONTROL

Access control is the process used to verify that the user who interacts with the application is authorized to perform the specific CRUD action (Create / Read / Update / Delete).

3.3 SESSION MANAGEMENT

The HTTP protocol used for the communication between the browser and the application is stateless by definition.

This means that HTTP treats each request as an independent transaction that is unrelated to any previous request so that the communication consists of independent pairs of request and response.

This property of the protocol makes necessary the creation of a mechanism that allows identifying the state of a user inside the application.

One solution consists in the use of session cookies or SessionIDs: for each request that the client sends to the server, the application returns this cookie so that the server is able to bind the requests to the session of the correct user.

Session identifiers must be unique, not predictable, and hard to reverse engineer. These aspects make important to use pseudorandom numbers generation algorithms that are cryptographically strong.

3.4 DATA PROTECTION - CRYPTOGRAPHY

Cryptography is the process of data protection. The purpose of this process is to make any sensitive data not understandable by a malicious user who has managed to intercept them.

It is important be sure that any sensitive data is protected by encryption performed through the use of standard algorithms whose robustness is proven.

3.5 AUDIT AND LOGGING

Indicates the set of activities related to the collection and analysis of log data collected in the application and system log messages.

3.6 SENSITIVE DATA MANAGEMENT

Indicates the management of sensitive data used by the application both at the server side and at the client-side level.

3.7 DATA VALIDATION

Data validation area represents the most complex part to protect. Indeed, every input and output parameter could involve critical vulnerabilities and attacks against final users.

It is fundamental to build an application that implements correct input data validation techniques and the encoding of the output data. This must check the correctness of the variables before interacting with the DB, web services, file system and operating system.

3.8 ERROR HANDLING

A correct errors handling process allows to report to the user during the interaction with the application. Error messages that contain too much information can lead to information disclosure about internal flow of the application, which could be used in a subsequent attack.

3.9 USE OF HTTP HEADERS IN APPLICATION SECURITY

The following HTTP headers have been proposed by W3G and implemented in browsers to improve user safety while browsing. Their use is strongly recommended unless explicit design choices due to operational aspects of the application itself.

- HTTP Strict Transport Security (HSTS)
- Public Key Pinning Extension for HTTP (HPKP)
- X-Frame-Options
- X-XSS-Protection
- X-Content-Type-Options
- Content-Security-Policy
- X-Permitted-Cross-Domain-Policies
- Referrer-Policy
- Expect-CT
- Feature-Policy

To learn more about the topic, refer to the following OWASP project:

- https://www.owasp.org/index.php/OWASP_Secure_Headers_Project

3.10 CONFIGURATION MANAGEMENT

This section presents some suggestions aimed at avoiding problems related to the configuration of the environment where web applications run rather than problems that may be present in the application code. For this reason, the suggestions listed may have specific solutions that depend on the specific environment in use.

4 REQUIREMENT SPECIFICATION

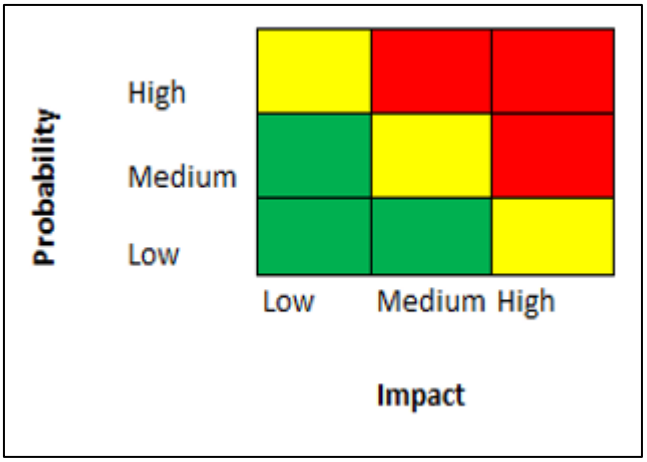
4.1 REQUIREMENT SPECIFICATION

For each requirement stated during the analysis activity, the following aspects were evaluated:

- **Difficulty of exploitation** by an attacker.
- **Technological impact (non-business)** in the event that the vulnerability is exploited by an attacker.
- **Difficulty of resolution** by applying the requirement requested by the customer.
- **Priority of intervention** in the introduction of the required safety requirement.

Based on the previous aspects, the risk in the event of a hypothetical vulnerability present in the system was taken into account for each requirement. This risk is given by the product of the **probability** of the occurrence of an attack due to the vulnerability and the **technological impact** from the exploitation of this activity.

The image below shows the **risk** calculation matrix:



Therefore, the proposed **priority of intervention** takes into account the **risk** value in case of vulnerabilities present in the system, due to the failure to adopt the required security requirements and the **difficulty** in satisfying these requirements, which is measured as the effort by the customer in applying all the countermeasures described within the proposed security requirements.

The table below shows the rules for using the terms used associated with the applicable priority values, in order to formalize the terminology within the security requirements:

Term	Category	Action
Is necessary Is mandatory	High/Medium	Mandatory
Is strongly suggested Is important to consider	Medium	Not mandatory but it is necessary to assess the risks in case of non-implementation
Is suggested It should be considered	Low	Implementation is not necessary except in situations of particularly stringent safety requirements

The terms: "It is strongly recommended / It is important to consider", indicate that the implementation choice is inherent to business aspects and internal risk analysis that a generic document such as this one cannot consider; therefore, the final implementation choice is left to the customer.

Finally, since the guidelines are a document that identifies and categorizes risk aspects without contextualizing specific applications, the end user can justify the failure to implement a specific control by taking the risk of this choice.

4.2 LIST OF SECURITY REQUIREMENTS

Code		Category	Name	Priority
RU1	4.3.1	Authentication	Send private information over encrypted channels	High
RU2	4.3.2	Authentication	Protect from User Enumeration	Medium
RU3	4.3.3	Authentication	Protection from guessable (dictionary) user account	Low
RU4	4.3.4	Authentication	Implement a strong password policy	High
RU5	4.3.5	Authentication	Avoid authentication bypass for private resources	High
RU6	4.3.6	Authentication	Implement a correct password reset method	High
RU7	4.3.7	Authentication	Avoid caching sensitive data	Medium
RU8	4.3.8	Authentication	Avoid positive authentication	High
RU9	4.3.9	Authentication	Remove application default accounts	High
RU10	4.3.10	Authentication	Verify wrong authentication attempts	Medium
RU11	4.3.11	Authentication	Implement a secure password change functionality	High
RU12	4.3.12	Authentication	Send private information via POST	Medium
RU13	4.3.13	Access Control	Protect from Path Traversal	High

RU14	4.3.14	Access Control	Protect resources from unauthorized access	**High**
RU15	4.3.15	Access Control	Avoid privilege escalation	**High**
RU16	4.3.16	Access Control	Avoid authorization bypass	**High**
RU17	4.3.17	Access Control	Correctly manage third party code	**Medium**
RU18	4.3.18	Access Control	Correctly handle Cross Origin (CORS) resources	**Medium**
RU19	4.3.19	Access Control	Implementing anti automation controls	**Low**
RU20	4.3.20	Access Control	Correct use of Websockets	**Medium**
RU21	4.3.21	Session Management	Build a secure HTTP session	**High**
RU22	4.3.22	Session Management	Set secure attributes for the session cookies	**High**
RU23	4.3.23	Session Management	Renew the HTTP session after successful login	**High**
RU24	4.3.24	Session Management	Protect from Cross Site Request Forgery	**High**
RU25	4.3.25	Session Management	Check the uniqueness of the user's session	**High**
RU26	4.3.26	Session Management	Isolate session keys	**High**
RU27	4.3.27	Session Management	Implement a correct logout functionality	**Medium**
RU28	4.3.28	Data protection - Encryption	Protect client/browser and server communication	**Medium**
RU29	4.3.29	Data protection - Encryption	Use standard cryptographic algorithms	**Medium**
RU30	4.3.30	Data protection - Encryption	Protect from Padding Oracle attacks	**Medium**
RU31	4.3.31	Data protection - Encryption	Secure Random Number implementation	**Medium**
RU32	4.3.32	Audit and Logging	Correctly manage security events	**Medium**
RU33	4.3.33	Audit and Logging	Protect log files	**High**

RU34 4.3.34	Validation of sensitive data	Prevent caching of sensitive data on client side	Medium
RU35 4.3.35	Validation of sensitive data	Validate user input	High
RU36 4.3.36	Validation of sensitive data	Output encoding	High
RU37 4.3.37	Validation of sensitive data	Use of prepared statements	High
RU38 4.3.38	Validation of sensitive data	Correctly build HTTP requests	High
RU39 4.3.39	Error Handling	Correctly handle application errors	Medium
RU40 4.3.40	Usage of HTTP headers on application security	Use the X-Frame-Options header	Low
RU41 4.3.41	Usage of HTTP headers on application security	Use the X-XSS-Protection header	Low
RU42 4.3.42	Usage of HTTP headers on application security	Use HTTP Strict Transport Security	Low
RU43 4.3.43	Usage of HTTP headers on application security	Use the Content-Security-Policy header	Low
RU44 4.3.44	Configuration Management	Protect the administrative interfaces	Medium
RU45 4.3.45	Configuration Management	Disable directory listing	High
RU46 4.3.46	Configuration Management	Disable dangerous HTTP methods	Low
RU47 4.3.47	Configuration Management	Remove system default accounts	Medium
RU48 4.3.48	Configuration Management	Remove unused files	Medium
RU49 4.3.49	Configuration Management	Protect from HTTP Verb Tampering	Medium

RU50 4.3.50	Configuration Management	Avoid use of private embedded data in HTML code	**Medium**
RU51 4.3.51	Configuration Management	Correctly configure extensions handling	**Medium**
RU52 4.3.52	Configuration Management	Use components without know vulnerabilities	**Medium**

4.3 REQUIREMENTS DESCRIPTION

Below is a detailed description of the required security requirements listed by application categories.

4.3.1 SEND PRIVATE INFORMATION OVER ENCRYPTED CHANNELS

Requirement ID	AUT-001
Priority	**High**
Description	Credentials or, in general, private data exchanged between client and server (such as authentication cookies), or between web application layers, should always travel over an encrypted channel. It is necessary to make sure that private data (e.g. usernames and passwords) are sent over an encrypted channel (HTTPS). Furthermore, only HTTP methods such as POST, PUT or PATH should be used to send credentials. Avoid using the GET method, as it could expose private information inside log files or browser history. The failure to use an encrypted channel exposes the application to Man In The Middle attacks, which could allow an attacker to intercept the traffic and acquire private information. **References**: • https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

Java/J2EE	To ensure that all pages are served via HTTPS, the web.xml configuration can be edited to make all the pages in the folder /web/secure accessible only via HTTPS, as shown below:
	`<security-constraint>`
	`  <web-resource-collection>`
	`  <web-resource-name>Security page</web-resource-name>`
	`  <url-pattern>/web/secure/*</url-pattern>`
	`  </web-resource-collection>`
	`  <user-data-constraint>`
	`  <transport-guarantee>CONFIDENTIAL</transport-guarantee>`
	`  </user-data-constraint>`
	`</security-constraint>`
	The same result can also be obtained via annotation, as in the following example:
	`@WebServlet("/web")`
	`@ServletSecurity(@HttpConstraint(transportGuarantee =`
	`TransportGuarantee.CONFIDENTIAL))`
	`public class MyServlet extends HttpServlet {`

	// servlet code...
	}
	
	References:
	• https://docs.oracle.com/javaee/7/tutorial/security-webtier002.htm

Java/Spring	In case Spring Security is used, to be sure that sensitive information travels on an encrypted channel it is possible to edit the Spring Security configuration file, by specifying the required protocol through the attribute requires-channel in the element <intercept-url>, as shown in the following example:
	
	<http auto-config="true" access-denied-page="/auth/denied">
	<intercept-url pattern="/admin/*" access="ROLE_ADMIN" requires-channel="https" />
	<intercept-url pattern="/user/*" access="ROLE_USER" requires-channel="https"/>
	<intercept-url pattern="/auth/login" access="IS_AUTHENTICATED_ANONYMOUSLY" requires-channel="https"/>
	<intercept-url pattern="/**" access="IS_AUTHENTICATED_ANONYMOUSLY" requires-channel="https"/>
	<form-login login-page="/auth/login" authentication-failure-url="/auth/login?login_error=true" default-target-url="/user"/>
	<logout logout-url="/auth/logout" logout-success-url="/" invalidate-session="true"/>
	</http>
	
	Spring also offers support for Java Configuration thus not requiring the creation of any XML files. In order to force an encrypted channel for all requests, it is possible to use a configuration as shown in the following extension of the WebSecurityConfigurerAdapter class:
	
	@Configuration
	@EnableWebSecurity
	public class SecurityConfig extends WebSecurityConfigurerAdapter {
	@Override
	protected void configure(HttpSecurity http) throws Exception {
	http
	.requiresChannel()
	.anyRequest().requiresSecure();
	// altre configurazioni
	}
	}

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Credentials_Transported_over_an_Encrypted_Channel_(OTG-AUTHN-001) • https://www.owasp.org/index.php/Testing_for_Sensitive_information_sent_via_unencrypted_channels_(OTG-CRYPST-003)

4.3.2 PROTECT FROM USER ENUMERATION

Requirement ID	AUT-002
Priority	**Medium**

Description	An attacker could be able to identify if a username is valid or not for the application by trying to interact with the authentication system.
	This happens if the application replies in two different ways if a username exists or not, regardless of the password.
	Therefore, an application is affected by this vulnerability if, given a valid username and a wrong password, the system replies with a message like the following:
	`Login failed for User foo: invalid password`
	Otherwise, it replies with the following message when the user does not exist on the system:
	`Login failed for User foo: invalid Account`
	Be sure that the application does not provide too many details during the authentication phase and always provide the same generic error message.
	In case of a non-existing user on platform, display always the following generic message:
	Wrong Credentials
	Make sure that the server does not use different response times depending on whether the user exists or not in authentication process, to prevent an attacker from inferring through this mechanism, even if unchanged error message is provided, and to make user enumeration.
	Finally, it is important to notice that the login functionality is not the only one that can be abused to enumerate users of the platform. Another function is, for example, password recovery.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Account_Enumeration_and_Guessable_User_Account_(OTG-IDENT-004) • https://www.owasp.org/index.php/Authentication_Cheat_Sheet#Authentication_and_Error_Messages

4.3.3 PROTECTION FROM GUESSABLE (DICTIONARY) USER ACCOUNT

Requirement ID	AUT-003
Priority	**Low**

Description	If the application allows using “simple” passwords (e.g. only numeric ones), then this could simplify brute forcing attacks on username values. Generally, it is always suggested to allow the user to choose his own username. If this is not possible, it is advisable to follow the recommendations reported below. Best practice recommends to: • Avoid creating simple and guessable usernames such as Name.Surname or numeric ones. • Avoid using public seeds as the username or something that can be easily reconstructed starting from other information like the tax code. Finally, it is suggested to implement a secure password policy in order to avoid brute bruteforce attacks when a valid account is detected (minimum length, containing numbers, letters, and special characters – see also 4.3.4).
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Account_Enumeration_and_Guessable_User_Account_(OTG-IDENT-004) • https://www.owasp.org/index.php/Testing_for_Weak_or_unenforced_username_policy_(OTG-IDENT-005)

4.3.4 IMPLEMENT A STRONG PASSWORD POLICY

Requirement ID	AUT-004
Priority	**High**
Description	Brute forcing of access credentials consists in trying to guess the password of a user for which the username is known. This kind of attack is performed by using automatic tools that, given a username, try to guess the password making several tries. A particular case of bruteforcing is the dictionary attack in which the passwords are retrieved starting from a list of words instead of all possible sequences of characters. The second method obviously would allow an attacker to reach the result in much less time. It is necessary to implement a check in the phase of password creation and subsequent modification that requires the user to choose a password that is strong: • Length> = 8 characters. • At least 1 number. • At least 1 lowercase letter. • At least 1 uppercase letter. • At least 1 special character. Following a sample regular expression (Regexp) implementing a strong password policy:

	^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[-/:-@\[-`]).{8, }$

Java/J2EE	The following code example checks that the password has a minimum length of 8 characters, contains at least one number, an alphabetic symbol, and a special character.
	`public static boolean checkPassword(String password) throws Exception {`
	`  boolean retval = false;`
	`  String regex = "^.*(?=.{8,})(?=.*\\d)(?=.*[a-za-zA-Z])(?=.*[!@#$^%&+=\\?\\.\\*\\-\\_]).*$";`
	`  retval = password.matches(regex);`
	`  return retval;`
	`}`
	It is worth considering that check can also be implemented by using the bean validation that is part of the J2EE since version 6.
	E.g.:
	`public class Utente {`
	`  @NotNull`
	`  @Pattern(regexp="^.*(?=.{8,})(?=.*\\d)(?=.*[a-za-zA-Z])(?=.*[!@#$^%&+=\\?\\.\\*\\-\\_]).*$")`
	`  private String password;`
	`  ...`
	`}`

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Weak_password_policy_(OTG-AUTHN-007) • https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet

4.3.5 AVOID AUTHENTICATION BYPASS FOR PRIVATE RESOURCES

Requirement ID	AUT-005
Priority	**High**
Description	It is mandatory to make sure that access to resources protected by authentication are not reachable via direct requests from the browser without being previously authenticated. For every private page that requires authentication the application must check that the visiting user is authenticated and has a valid session. Otherwise, it is necessary to redirect him to the login page.

	If the access control is left to a platform specific implementation, it is necessary to make sure that it verifies the authentication status of the user for each protected resource.
Java/J2EE	The aforementioned best practices are valid also for these languages.

Java/Spring	Using Spring Security, it's possible to ensure that all requests are authenticated using a WebSecurityConfig as shown below:
	`protected void configure(HttpSecurity http) throws Exception {`
	`  http`
	`  .authorizeRequests().anyRequest().authenticated()`
	`  .and()`
	`  .formLogin().loginPage("/login").permitAll();`
	`}`
	
	Otherwise, the following XML configuration can be used:
	`<http>`
	`  <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>`
	`  <intercept-url pattern="/**" access="authenticated"/>`
	`  <form-login login-page='/login.jsp' />`
	`</http>`

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Bypassing_Authentication_Schema_(OTG-AUTHN-004)

4.3.6 IMPLEMENT A CORRECT PASSWORD RESET METHOD

Requirement ID	AUT-006
Priority	**High**
Description	Password reset systems allow a user to retrieve a forgotten password. It is important to pay attention to these systems since they are potentially weak, in particular when they are question based. Most of the time, indeed, the answers to the questions can be found searching on the web or using social engineering techniques. It is advisable to not use such functions in sensitive systems that contain private data. The following password reset flow is suggested: • The user access to the Password Reset functionality.

	• The application sends an e-mail to the user: this contains a onetime secret token to copy and paste that is valid only for a short amount of time. • If the user has not requested the reset, give him the possibility to notify the abuse. • Before password's reset, check the token expiration date and if it is linked with the proper user. • If the reset is successful, send a notification e-mail. It is advisable to log all the procedure server side. When the password reset method is question based, make sure that the questions are created by the user himself. The user should create at least 5 questions and 3 of them should be presented during the password reset phase. Warn the user to not insert questions that could be easily reconstructed searching on Internet, such as "what is your tax number?". As specified above refers to an SFA (Single-factor authentication) authentication, for 2FA (Two-factor authentication) refer to external guides in order to be aligned to the involved Identity Provider policies.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_weak_password_change_or_reset_functionalities_(OTG-AUTHN-009) • https://www.owasp.org/index.php/Testing_for_Weak_security_question/answer_(OTG-AUTHN-008) • https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet

4.3.7 AVOID CACHING SENSITIVE DATA

Requirement ID	AUT-007
Priority	**Medium**
Description	The caching mechanisms are generally used in order to improve the performance and speed of client-server communication in case of exchange of static information or data shared several times. However, these mechanisms risk allowing the caching of sensitive information returned by the web pages of applications, such as: • personal profiles. • pages with private images. allowing in fact the saving in the browser used for navigation of the displayed data. For example, an attacker with physical access to the device used for browsing could use the "back" button of the browser and navigate backwards through the history, reading and

	viewing the pages that the web application has allowed to cache on the client in previous browsing sessions of other users.
	In order to ensure the greatest possible coverage even by old devices that support version 1.0 of HTTP, it is necessary to implement the following set of directives within the web server's HTTP responses:
	`Cache-Control: no-cache, no-store, must-revalidate`
	`Pragma: no-cache`
	`Expires: 0`
	References:
	• https://tools.ietf.org/html/rfc7234#section-5.2

Java/J2EE	Below is shown an example of how to programmatically set the header inside a filter:
	`public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {`
	`  HttpServletResponse res = (HttpServletResponse)response;`
	`  res.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");`
	`  res.setHeader("Pragma", "no-cache");`
	`  res.setHeader("Expires", "0");`
	`  chain.doFilter(request, response);`
	`}`
	This filter will then be added to the web.xml in order to be valid for all pages, as showed below:
	`<filter>`
	`  <filter-name>CacheControlFilter</filter-name>`
	`  <filter-class>com.org.CacheControlFilter</filter-class>`
	`</filter>`
	`<filter-mapping>`
	`  <filter-name>CacheControlFilter</filter-name>`
	`  <url-pattern>/*</url-pattern>`
	`</filter-mapping>`

Java/Spring	Spring Security adds by default some headers in order to improve the application security.
	For example, these headers are added by default in each response:
	`Cache-Control: no-cache, no-store, max-age=0, must-revalidate`
	`Pragma: no-cache`
	`Expires: 0`

	Below is an example of how to configure these headers:
	
	`<http>`
	`  <!-- ... -->`
	`  <headers defaults-disable="true">`
	`  <cache-control />`
	`  </headers>`
	`</http>`

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Browser_cache_weakness_(OTG-AUTHN-006)

4.3.8 AVOID POSITIVE AUTHENTICATION

Requirement ID	AUT-008
Priority	**High**
Description	In case of failure of the login functionality, it is necessary to guarantee that the default user is not used as fallback. This issue could happen when the fail rate of the login function is considered low, it is then necessary to always use a negative authentication when there are errors during the login process. Specifically, in case of errors, the authentication should always be considered unsuccessful.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Bypassing_Authentication_Schema_(OTG-AUTHN-004)

4.3.9 REMOVE APPLICATION DEFAULT ACCOUNTS

Requirement ID	AUT-009
Priority	**High**
Description	Make sure that the applications do not support or use default accounts installed by the running systems since they could be easily identified by an attacker. Furthermore, it is necessary that all accounts, especially the administrative ones, respect the password policy discussed in the dedicated paragraph.

	A malicious user could exploit default credentials to access authenticated features and gain greater attack surface.
Reference for verification	https://www.owasp.org/index.php/Testing_for_default_credentials_(OTG-AUTHN-002)

4.3.10 VERIFY WRONG AUTHENTICATION ATTEMPTS

Requirement ID	AUT-010
Priority	**Medium**
Description	Allowing an infinite number of login attempts implies the possibility for an attacker to be able to carry out bruteforcing attacks, thus allowing him to identify the correct credentials. The authentication process must disable a user account if the user has inserted a wrong password for many times. Make sure that the server does not take different times to respond to authentication depending on whether the authentication succeeded or went wrong, to prevent an attacker from inferring information through this mechanism. Alternative Solution: After the third wrong login attempt, gradually slow down the server response and after the third failed attempt ask for a CAPTCHA resolution. Show an appropriate message to the user during the wait.

Java/Spring	One way to make this control in Spring is to implement and register org.springframework.context.ApplicationListener interface. Then inside the listener, it is possible to check the returned event by the login attempt.
	`public class ApplicationEventListener implements`
	`  org.springframework.context.ApplicationListener {`
	`  public void onApplicationEvent(ApplicationEvent event) {`
	`  if (event instanceof AuthenticationFailureBadCredentialsEvent){`
	`  // handle the login event failed`
	`  // (increase the retry counter)`
	`  }else if (event instanceof AuthenticationSuccessEvent) {`
	`  // manage the successful login event`
	`  // reset the attempts counter for the user`
	`  }`
	`  }`
	`}`

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Weak_lock_out_mechanism_(OTG-AUTHN-003)

4.3.11 IMPLEMENT A SECURE PASSWORD CHANGE FUNCTIONALITY

Requirement ID	AUT-011
Priority	**High**
Description	The password change process must take place in the authenticated part of the application, after successful user authentication. The change password feature must ask for the old password and the new password 2 times in the same form. This allows to prevent possible vulnerabilities such as *Cross Site Request Forgery*: • *https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)* or *user enumeration*: • *https://www.owasp.org/index.php/Testing_for_User_Enumeration_and_Guessable_User_Account_(OWASP-AT-002)* Before changing the password, the system must verify that the old password is the correct one and that the two new passwords have the same value. In the event that concurrent sessions are used, once the password has been changed, it is advisable to invalidate all active sessions related to the user whose password has been changed.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_weak_password_change_or_reset_functionalities_(OTG-AUTHN-009)

4.3.12 SEND PRIVATE INFORMATION VIA POST

Requirement ID	AUT-012
Priority	**Medium**
Description	Avoid the use of methods such as GET that can expose sensitive information transported in log files or in the browser history.

	When using the HTTP protocol, make sure that sensitive or confidential information (such as username or password) is sent using HTTP methods such as POST, PUT or PATCH. Sensitive information that is exchanged between client and server or through the various levels of a web application must not be exposed in clear text in the logs, browser history and more. Anyone with access to the logs etc., will be able to acquire the stored data and reuse them to carry out user impersonation attacks.

Java/Spring	Spring Framework provides the @RequestMapping annotation through which it is possible to indicate the handler which should be delegate, for example to authenticate, and specify the method that must be used to recall it.
	`@RequestMapping(value="/Login", method={RequestMethod.POST})`
	`public String handleLogin(){`
	`..`
	`// perform user login`
	`..`
	`}`
	This ensures that login operations can only be performed using the POST method to exchange credentials between client and server.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Credentials_Transported_over_an_Encrypted_Channel_(OTG-AUTHN-001)

4.3.13 PROTECT FROM PATH TRAVERSAL

Requirement ID	AZ-001
Priority	**High**

Description	Applications that implement access functionalities to resources such as pdf, xls, html, must adequately protect themselves from potential path traversal issues.
	For example, when the access to a resource is made as the following one:
	`http://www.esempio.it/leggifile.jsp?file=report.pdf`
	It is necessary to protect the application from a request like this:
	`http://www.esempio.it/leggifile.jsp?file=../../../../some dir/some_file`

	Failing to implement the correct checks could allow an attacker to access arbitrary files on the application's filesystem. It is necessary to approach these cases by keeping the files to be displayed in a specific folder of the file system, possibly not shared, whose path is well defined. Make sure that the path constructed to access the file matches the canonical one for the file. If this is not possible (for example if the application displays a series of documents to users) make sure that the user cannot enter the entire path of the file to be viewed. The best method is to make a white list of the strings that can be entered by the user. Validate user input by discarding strings that contain at least the "\", "/" or ".." characters. Finally, a possible alternative to the direct use of filesystems is to save content directly in a database and then apply the best practices inherent to the specific context to avoid problems related to SQL Injection.

Java/J2EE	Path traversal is an issue that can be prevented by using data validation and encoding.
	Data Validation:
	Identify the characters that the filename can contain. For example, check the filename (alphanumeric characters and underscore are admissible) and its extension:
	`if(!filename.matches("^[ a-z0-9-_]+\\.[a-z0-9]+$"){`
	`  throw SecurityException;`
	`}`
	If it is necessary to insert one or more directories in the path, it is possible to use multiple arguments representing a directory, avoiding allowing the '/' or '\' characters (UNIX/Windows).
	In this way, every argument will be validated with/^[a-z0-9-_]+$/
	If the file is saved on a remote server, it is important to check its extension using a whitelist approach.
	The second step is about the encoding of the file system access mode. Check if the absolute path file is expected using a control like the following one:
	`try{`
	`  String abs=file.getCanonicalFile().getParent()`
	`  if("/path/expected/the/file/is/".equals(abs)==false)`
	`  throw SecurityException;`
	`}catch(IOException exc){`
	`  ..`
	`}`

	Finally, it is recommended to check if the file will be accessed/written to the expected directory, by normalizing it to full path and comparing it to the expected full path folder.
Reference for verification	• https://www.owasp.org/index.php/Testing_Directory_traversal/file_include_(OTG-AUTHZ-001)

4.3.14 PROTECT RESOURCES FROM UNAUTHORIZED ACCESS

Requirement ID	AZ-002
Priority	**High**
Description	For every specific role inside the application, it is necessary to implement an authorization control related to the resources. This can avoid the case of users that access resources to which they are not allowed to. Furthermore, make sure that non administrative users cannot access resources reserved only to the administrators: save the accessible URLs with an administrative user and try to access them with a user that has lower rights. It is advisable to always check that the account number passed as value inside a request is linked to the session of the user that is doing the request. It is necessary to implement a control such that a user can access only the information related to his own checking account. A simple test is to memorize the URLs accessible with an administrator user and then try to access them with a user with lower permissions.

Java/J2EE	In addition to custom solutions, some of the J2EE native methods or frameworks can be used.
	Regarding J2EE, the method is: isUserInRole() which allows to perform the check based on the internal role.
	For example, assuming that the functionality is accessible only by an administrative user of role “Admin”:
	`doRequest(ServletRequest req, ServletResponse res,`
	`FilterChain chain){`
	`if(!request.isUserInRole(“Admin”)){`
	`throw SecurityException;`
	`}else{`
	`// User is authorized, go on.`
	`}`

	If the resources that require a certain role are all in the same folder, you can also use the controls in configuration to block access to entire directories or resources through the "auth-constraint" element.
	The following example requires the server to check if user's roles is "manager" or "assistant" to access resources in /orders/ and that the connection is on https:
	`<security-constraint>`
	`  <web-resource-collection>`
	`  <web-resource-name>SecureOrders</web-resource-name>`
	`  <url-pattern>/orders/*</url-pattern>`
	`  </web-resource-collection>`
	`  <auth-constraint>`
	`  <role-name>assistant</role-name>`
	`  <role-name>manager</role-name>`
	`  </auth-constraint>`
	`  <user-data-constraint>`
	`  <transport-guarantee>CONFIDENTIAL</transport-guarantee>`
	`  </user-data-constraint>`
	`</security-constraint>`

Java/Spring	To protect access to certain resources in Spring Security, WebSecurityConfig can be used as shown below, where some pages are accessible to anyone while everything in admin will be accessible to users of ADMIN role. All other requests generally require authentication:
	`protected void configure(HttpSecurity http) throws Exception {`
	`  http`
	`  .authorizeRequests()`
	`  .antMatchers("/resources/**", "/signup", "/about").permitAll()`
	`  .antMatchers("/admin/**").hasRole("ADMIN")`
	`  .anyRequest().authenticated()`
	`  .and()`
	`  // ...`
	`  .formLogin();`
	`}`
	The same configuration can be done via XML file.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Bypassing_Authorization_Schema_(OTG-AUTHZ-002) • https://www.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_(OTG-AUTHZ-004)

4.3.15 AVOID PRIVILEGE ESCALATION

Requirement ID	AZ-003
Priority	**High**
Description	It is necessary to implement the resources access model such that the authorization logic resides only on the server part avoiding providing user's role information inside HTTP request variables. For example, if a user executes the following request to see an order: POST /utente/vedOrdine.jsp HTTP/1.1 Host: www.example.com ... gruppoID=grp001&ordineID=0001 In case the application verifies the user membership to a group by using the groupID parameter, it is important to note that such value is controllable client side and so it is subjected to the described issue.
Java/J2EE	Do not store information about roles or privileges on client-side. The role to which a user belongs must not be specified in the request but instead must be set in session at the login time.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Privilege_escalation_(OTG-AUTHZ-003)

4.3.16 AVOID AUTHORIZATION BYPASS

Requirement ID	AZ-004
Priority	**High**
Description	When there are functions that need a specific workflow in order to be authorized (e.g. Insert PIN1, then PIN2), it must not be possible to directly access the final step without inserting the PIN1 and PIN2 before. If this happens, an authorization bypass occurs.
Java/J2EE	It is advisable to create an application state that defines if the insertion of PIN1 and PIN2 has been successfully done; this state will have to be checked in the next requests. It is important to correctly analyze the execution flow of the dispositive operations. For example, to set the new password, it must be present the check of the old one.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Bypassing_Authorization_Schema_(OTG-AUTHZ-002) • https://www.owasp.org/index.php/Testing_for_the_Circumvention_of_Work_Flows_(OTG-BUSLOGIC-006)

4.3.17 CORRECTLY MANAGE THIRD PARTY CODE

Requirement ID	AZ-005
Priority	**Medium**
Description	The use of third-party code (e.g. JavaScript) could allow the third party to execute arbitrary code on the users' browsers. A common attack case includes credential theft or session cookie theft. It is advisable to not dynamically load third party contents but import all the JavaScript files directly on the web site. It is advisable to check all the external scripts that are loaded on the users' browsers. It's suggested to check all external scripts that are loaded in the users' browsers, to identify which one have access to the applications and to find out if sensitive information is collected and where it is sent. In this case, if all the important cookies such as "HttpOnly" are set it is possible to avoid that a script can access cookies set on the user's browser.

Java/J2EE	It is advisable to import third party JavaScript code directly on the frontend webserver after a careful review.
	Following an example of a secure import:
	`<SCRIPT Language=Javascript src="trackingcookie.js">`
	It is advisable to not include JavaScript code from third party sites since it can be possible for an attacker to execute an attack on the current site, compromising the application that hosts the included JavaScript code.
	Potentially **insecure** example:
	`<SCRIPT Language=Javascript src="https://example.com/trackingcookie.js">`
	A safety analysis of the third-party site (example.com) is fundamental if only the above solution can be adopted.
	Only if it is not possible to locally mirror the third-party JavaScript libraries, use the "integrity" attribute for the script tag to ensure that the version of the external library has not been modified by a malicious actor.

	<script src="https://analytics.vendor.com/v1.1/script.js" integrity="sha384-MBO5IDfYaE6c6Aao94oZrIOiC7CGiSNE64QUbHNPhzk8Xhm0djE6QqTpL0HzTUxk" crossorigin="anonymous"></script> However, keep in mind that this attribute is not yet supported by all modern browsers and server that exposes these libraries must accept CORS requests. • https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
Reference for verification	• https://www.owasp.org/index.php/3rd_Party_Javascript_Management_Cheat_Sheet • https://www.owasp.org/index.php/Test_business_logic_data_validation_(OTG-BUSLOGIC-001)

4.3.18 CORRECTLY HANDLE CROSS ORIGIN (CORS) RESOURCES

Requirement ID	AZ-006
Priority	**Medium**
Description	When a read access on a resource is required by a Flash content or by a CORS request, it is necessary to define a list of allowed sites that can access the requested resource. In particular, for CORS: Make sure that the URLs that reply with: Access-Control-Allow-Origin: * Or with: Access-Control-Allow-Origin: [VALUE-FROM-ORIGIN-REQUEST] Or that simply add the referrer name to the Access-Control-Allow-Origin header, do not include the sensitive content. Use Access-Control-Allow-Origin only on a restricted number of URLs and perform the control on the Origin header checking if it corresponds to a trusted host. It is worth considering that CORS headers do not block the request but only prevent reading by the browser. Therefore, it is important to execute the anti CSRF mitigation in any case.

Java/Spring

Starting from the Spring Framework version 4.2, CORS is natively supported. By default, Spring does not place constraints on the domains from which requests may arise nor on the methods that can be used to make CORS requests.

It is possible to enable a controller for CORS requests through the appropriate @CrossOrigin annotation. For example, the following will only accept CORS requests from the domain2.com domain:

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {
  @RequestMapping("/{id}")
  public Account retrieve(@PathVariable Long id) {
        // ...
  }
}
```

CORS can also be configured at the application level, rather than on every single controller, for example through the WebConfig. The following example shown how to specify the source domain, methods, and that credentials are not required.

Spring4 example:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
      .allowedOrigins("http://domain2.com")
      .allowedMethods("PUT", "DELETE")
      .allowCredentials(false).maxAge(3600);
        }
}
```

Spring5 example:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
      .allowedOrigins("http://domain2.com")
      .allowedMethods("PUT", "DELETE")
      .allowCredentials(false).maxAge(3600);
        }
}
```

	If Spring Security is used, the CORS support can be enabled in this way:
	`@EnableWebSecurity`
	`public class WebSecurityConfig extends WebSecurityConfigurerAdapter {`
	
	`@Override`
	`protected void configure(HttpSecurity http){`
	`http.cors();`
	`}`
	
	`@Bean`
	`CorsConfigurationSource corsConfigurationSource() {`
	`CorsConfiguration config = new CorsConfiguration();`
	`config.setAllowedOrigins(Arrays.asList("https://example.com"));`
	`config.setAllowedMethods(Arrays.asList("GET","POST"));`
	`UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();`
	`source.registerCorsConfiguration("/**", config);`
	`return source;`
	`}`
	`}`
	
	Similarly, the same configuration can be done in XML format.

Reference for verification	• https://www.owasp.org/index.php/Test_Cross_Origin_Resource_Sharing_(OTG-CLIENT-007) • https://www.owasp.org/index.php/Test_RIA_cross_domain_policy_(OTG-CONFIG-008)

4.3.19 IMPLEMENTING ANTI AUTOMATION CONTROLS

Requirement ID	AZ-007
Priority	**Low**
Description	If it is necessary to implement a functionality that consists in adding data via form (like contact request or the users' registration functionality), it is important to implement a system that limits the automatic use of such functionalities. It is in fact possible that a malicious user automatically executes multiple requests that could fill the DB with data or create unexpected excesses of data. It is therefore strongly recommended to add so-called CAPTCHA systems or throttling systems that are capable of mitigating these types of attacks.

	Furthermore, if there is a WAF for blocking bots, it is recommended to configure it to block connections not only via the IP but also via a fingerprint, such as in the case of bruteforcing: `POST https://www.host.com/path/name/login user=XXXX&pass=????` In any case, please note that the unilateral blocking of the IP can lead to service continuity problems for legitimate users who use the same IP in NAT mode.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Weak_lock_out_mechanism_(OTG-AUTHN-003) • https://www.owasp.org/index.php/Test_User_Registration_Process_(OTG-IDENT-002) • https://www.owasp.org/index.php/Testing_for_Captcha_(OWASP-AT-012)

4.3.20 CORRECT USE OF WEBSOCKETS

Requirement ID	AZ-008
Priority	**Medium**
Description	WebSockets are a technology that makes it possible to open an interactive communication session between the user's browser and a server. Applications are able, through this API, to send messages to the server and receive event-driven responses without connection and preamble overhead for each request. Since WebSockets: • are not bound by the same original policy and do not perform CORS checks. • do not need a response header from the server to authorize the action. From a security perspective, it should be considered as a low-level communication over HTTP, aside from the possible authentication of the user on the web server. The server must take precautions to carry out all the authorization checks necessary to allow the connection by the browser. Then: • Check for "Origin" header value which contains the origin of the page that is attempting to communicate. • Check for the security of the transmission channel, especially if session cookies or other sensitive data are exchanged. • Take care of any user input and validate and encode it to prevent injection issues.

<table>
<tr><td></td><td>

- If the client side accepts user input, take care of any user input and validate and encode it to prevent injection issues.

It is also recommended by the server to expose the protocol version described in RFC 6455 because it is a more recent and secure version than previous versions and is still supported by all modern browsers.
</td></tr>
<tr><td>Java/J2EE</td><td>

J2EE 7 supports WebSockets through the Java API for WebSocket (JSR 356).

However, the controls and considerations made in the previous section remain the responsibility of the developers.
</td></tr>
<tr><td>Java/Spring</td><td>

Spring Security helps to create secure WebSockets starting since version 4. In fact, authentication and authorization checks can be requested on WebSocket handlers through a configuration as shown below:

```
@Configuration
public class WebSocketSecurityConfig extends
AbstractSecurityWebSocketMessageBrokerConfigurer {

  @Override
  protected void configureInbound(MessageSecurityMetadataSourceRegistry
messages) {
    messages
      .nullDestMatcher().authenticated()
      .simpSubscribeDestMatchers("/user/queue/errors").permitAll()
      .simpDestMatchers("/app/**").hasRole("USER")
      .simpSubscribeDestMatchers("/user/**", "/topic/friends/*").hasRole("USER")
      .anyMessage().denyAll();
    }
}
```

This configuration requires that every message without destination be authenticated, anyone can subscribe to /user/queue/errors, the subscribe messages to the other two handlers also require the same role and any other message will be rejected.

The same configuration can also be specified with XML file.

To authenticate requests, WebSocket uses the same information found in HTTP requests when the connection is created. So, if a user is authenticated to the HTTP application then his WebSocket requests will also be authenticated.

Moreover, by default from the Spring Framework version 4.1.5, WebSocket messages are accepted only if they come from the same domain. This ensures Same Origin Policy also for WebSockets.

References:
</td></tr>
</table>

	• http://docs.spring.io/spring-security/site/docs/4.2.3.RELEASE/reference/htmlsingle/#websocket
Reference for verification	• https://www.owasp.org/index.php/Testing_WebSockets_(OTG-CLIENT-010) • https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#WebSockets

4.3.21 BUILD A SECURE HTTP SESSION

Requirement ID	SM-001
Priority	**High**
Description	It is advisable to use cookies as a mechanism to handle the session. It is preferable to not send session information inside GET requests; use session cookies for this purpose. Each user cookie shall comply the following properties: • *Unpredictability*: it shouldn't be possible for a user to understand the cookie generation mechanism. • *Uniqueness*: a new cookie must be generated after every user's authentication. • *Temporary validity:* the application must set a session timeout of about 10 minutes. • *Minimum length*: at least 30 characters. In this way, a malicious user will not be able to predict the values of these tokens with the purpose of impersonating the user connected to the predicted token.

Java/J2EE	With Servlet 3.0 specification, it was introduced the possibility to disable URL rewriting that is the responsible to add the JSESSIONID into the URLs, thus making it sent via GET.
	This mode is supported by default in order to make application backward compatibility to user agents that do not support cookies.
	Since modern browsers support this technology, it is advisable to disable URL rewriting through a configuration like the following to forces cookies to be used exclusively:
	`<web-app>`
	`  <session-config>`
	`  <tracking-mode>COOKIE</tracking-mode>`
	`  </session-config>`
	`</web-app>`
	Or programmatically with a call like the following:

	servletContext.setSessionTrackingModes(EnumSet.of(SessionTrackingMode.cookie)); It should be considered that all these are valid for the application servers that implement the Servlet 3.0 specification. **References**: • http://docs.oracle.com/javaee/7/api/index.html?javax/servlet/annotation/ServletSecurity.html
Java/Spring	Spring Security handles this behavior through the disable-url-rewriting HTTP element attribute: <http disable-url-rewriting="true"> ... </http> For the sake of completeness, it is worth noting that since Spring Security version 4, the default value of this attribute is already set to "true". **References**: • http://docs.spring.io/spring-security/site/migrate/current/3-to-4/html5/migrate-3-to-4-xml.html
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Session_Management_Schema_(OTG-SESS-001) • https://www.owasp.org/index.php/Testing_for_Exposed_Session_Variables_(OTG-SESS-004) • https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

4.3.22 SET SECURE ATTRIBUTES FOR THE SESSION COOKIES

Requirement ID	SM-002
Priority	**High**
Description	Cookies are often a key attack vector for malicious users (typically targeting other users) and the application should always protect them by setting the following attributes for each cookie: • **Secure** (it indicates that the cookie can be exchanged only over secure channels like HTTPS. This avoids that the cookie can transit over clear connections. Note

	that if you use this attribute, you have to implement the HTTPS protocol within all the application; otherwise, you will encounter session related problems). • **HttpOnly** (it indicates that the cookie is not accessible by client-side scripting languages like JavaScript. This helps to prevent attacks like XSS that can be used to steal cookies). • **Domain** (it indicates for which domain it is necessary to send the cookie when the requests are sent. Obviously, setting a too wide value for this attribute means that the cookie will be sent more often). • **Path** (it specifies the URL path for which the cookie is valid. If domain and path are equals, then the cookie is sent by the client). • **Expires** (it indicates the cookie expiration date. It is advisable to make sure that the cookie does not expire too far and to handle the expiration server side). The use of these flags allows mitigating client-side attacks such as XSS or the reuse of cookies.

Java/J2EE	Java2EE supports HttpOnly and Secure since version 6 (Servlet class version 3):
	• http://java.sun.com/javaee/6/docs/api/javax/servlet/http/Cookie.html#setHttpOnly(boolean)
	Also, for session cookies (JSESSIONID)
	• http://java.sun.com/javaee/6/docs/api/javax/servlet/SessionCookieConfig.html#setHttpOnly(boolean)
	For older versions, it is possible to use a solution in code by rewriting the JSESSIONID by setting a custom header.
	`String sessionid = request.getSession().getId();`
	`response.setHeader("SET-COOKIE", "JSESSIONID=" + sessionid + "; secure; HttpOnly");`

Reference for verification	• https://www.owasp.org/index.php/Testing_for_cookies_attributes_(OTG-SESS-002) • https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

4.3.23 RENEW THE HTTP SESSION AFTER SUCCESSFUL LOGIN

Requirement ID	SM-003
Priority	**High**
Description	Session fixation vulnerability happens when the application does not set a new cookie after the user has been successfully authenticated.

	In this case the attack consists of obtaining a valid session ID (e.g. by connecting to the application), inducing a user to authenticate himself with that session ID, and then hijacking the user-validated session by the knowledge of the used session ID. The application must always set a new session cookie after the user has been successfully authenticated.

Java/J2EE	Servlet 3.1 provides the HttpServletRequest.changeSessionId() method that changes the session identifier without losing its attributes.
	`public void doLogin(HttpServletRequest request) {`
	`  request.changeSessionId();`
	`  authenticate(request); // username/password checked here`
	`}`
	Below, there is a generic example where the old session is invalidated and a new one is created when credentials are checked (without losing the old session attributes content):
	`public void doLogin(HttpServletRequest request) {`
	`  HttpSession oldSession = request.getSession(false);`
	`  if (oldSession != null) {`
	`  // create new session if there was an old session`
	`  HttpSession newSession = request.getSession(true);`
	`  Enumeration enumeration = oldSession.getAttributeNames();`
	`  while (enumeration.hasMoreElements()) {`
	`  String name = (String) enumeration.nextElement();`
	`  Object obj = oldSession.getAttribute(name);`
	`  newSession.setAttribute(name, obj);`
	`  }`
	`  oldSession.invalidate();`
	`  }`
	`authenticate(request); // username/password checked here`
	`}`

Java/Spring	By default, Spring Security prevents Session Fixation issues by renewing the session identifier value. This protection is active by default and does not need to be specifically enabled.
	The default enabled configuration is as follows:
	`<session-management session-fixation-protection="migrateSession">`
	Otherwise, it can be done via Java configuration:
	`@Configuration`
	`@EnableWebSecurity`
	`public class SecurityConfig extends WebSecurityConfigurerAdapter {`
	`  @Override`

	protected void configure(HttpSecurity http) throws Exception {
	http.sessionManagement()
	.sessionFixation().migrateSession();
	}
	}
	
	This way, Spring Security will generate a new session upon login and will retain the attributes of the old one.
	
	References:
	• http://docs.spring.io/spring-security/site/docs/4.2.3.RELEASE/apidocs/org/springframework/security/config/annotation/web/configurers/SessionManagementConfigurer.html

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Session_Fixation_(OTG-SESS-003) • https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

4.3.24 PROTECT FROM CROSS SITE REQUEST FORGERY

Requirement ID	SM-004
Priority	**High**
Description	Each application function that involves a change in the user's status (e.g. password change, modification/cancellation of personal data, administrative functions, etc.) must require a unique parameter for the single operation or require a dispositive password to protect against possible Cross Site Request Forgery attacks. The most common method to prevent these attacks is the Synchronizer Token. **References**: • https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Java/J2EE	When user successful make request for a page containing a form whose action modifies a state, create a cryptographically secure random token, and save it in session:
	
	private String createToken(Session session){
	String token= java.util.UUID.randomUUID().toString();
	session.setAttribute(“token”,token);
	}

	Insert the token in the form as a hidden field:
	`<input type=”hidden” name=”token” value=”${token}”/>`
	
	Finally, when the request is handled by the controller, it will first have to take care of checking the token presence in request and then the equality with the one stored in session.
	`Public void doPost(HttpServletRequest request, HttpServletResponse response) {`
	
	`if(!request.getSession().getAttribute(“token”).equals(request.getParameter(“token”))){`
	`  throw SecurityException;`
	`  }else{`
	`  // Do action`
	`  }`
	`}`

Java/Spring	Spring Security provides a CSRF attack protection, which implements the Token Pattern Synchronizer.
	The following example show how to enable CSRF protection via XML file:
	`<http>`
	`  <!-- ... -->`
	`  <csrf />`
	`</http>`
	
	The protection from the CSRF is enabled by default if Java Configuration is used.
	It is worth noting that since version 4 of Spring Security, the protection against CSRF is enabled by default even when using XML configuration, so there is no need to specify the relative element.
	The next step is to include the CSRF token in all those requests that we want to protect (POST, PUT, PATCH, DELETE) through, for example, the _csrf attribute:
	`<c:url var="logoutUrl" value="/logout"/>`
	`<form action="${logoutUrl}" method="post">`
	`<input type="submit" value="Log out" />`
	`<input type="hidden" name="${_csrf.parameterName}"  value="${_csrf.token}"/>`
	`</form>`
	
	In case is needed to protect AJAX or JSON requests, the CSRF token can be included in a custom header whose name and value can be exchanged via meta tag:

	<html> <head> <meta name="_csrf" content="${_csrf.token}"/> <!-- default header name is X-CSRF-TOKEN --> <meta name="_csrf_header" content="${_csrf.headerName}"/> <!-- ... --> </head> <!-- ... --> These values will then be retrieved via JavaScript to be added to all requests. To ensure integration with AngularJS, Spring Security implements the Double Submit Cookie through the CookieCsrfTokenRepository which will write the anti CSRF token value inside the XSRF-TOKEN cookie (or the _csrf attribute) and then read it from the custom header X-XSRF-TOKEN: <http> <!-- ... --> <csrf token-repository-ref="tokenRepository"/> </http> <b:bean id="tokenRepository" class="org.springframework.security.web.csrf.CookieCsrfTokenRepository" p:cookieHttpOnly="false"/> **References**: • http://docs.spring.io/spring-security/site/migrate/current/3-to-4/html5/migrate-3-to-4-xml.html • http://docs.spring.io/spring-security/site/docs/4.2.3.RELEASE/reference/htmlsingle/#csrf
Reference for verification	• https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005)

4.3.25 CHECK THE UNIQUENESS OF THE USER'S SESSION

Requirement ID	SM-005
Priority	**High**
Description	For critical applications, implement a check that verifies if an authenticated user can open a new session when the active one is not expired yet. If this is possible, keep track of the event.

	Also, make sure that the user IP is correlated to his session when accesses application resources.

Java/J2EE	Add the source IP in the session at login time.
	`request.getSession().setAttribute("IP",request.getRemoteAddr());`
	
	After that, insert a checking filter to verify that the IP that originated the request is equal to that saved in session.
	`public void doFilter(ServletRequest req, ServletResponse res,`
	`FilterChain chain){`
	`if(req.getSession()!=null && req.getSession().getAttribute("IP") != null &&`
	`!req.getSession().getAttribute("IP").equals(req.getRemoteAddr()))`
	`// potential session hijacking, throw loginException`
	`else`
	`// go on with filtering`
	`}`

Java/Spring	Spring Security allows controlling how many concurrent sessions are allowed through the concurrency-control element. By default, Spring does not limit the number of sessions that can be opened with the same credentials. The example below shows how to setup it with XML file:
	`<http ...>`
	`<session-management>`
	`<concurrency-control max-sessions="1" />`
	`</session-management>`
	`</http>`
	
	Otherwise, Java configuration can be used as shown below:
	`@Override`
	`protected void configure(HttpSecurity http) throws Exception {`
	`http.sessionManagement().maximumSessions(1);`
	`}`
	
	While using this feature, must be sure the HttpSessionEventPublisher is configured to ensure that the Spring Security session registry is notified when a session is destroyed.
	`<listener>`
	`<listener-class>`
	`org.springframework.security.web.session.HttpSessionEventPublisher`
	`</listener-class>`

	</listener>
Reference for verification	• https://www.owasp.org/index.php/Session_Management_Cheat_Sheet#Simultaneous_Session_Logons

4.3.26 ISOLATE SESSION KEYS

Requirement ID	SM-006
Priority	**High**
Description	It is necessary to avoid a correspondence between session keys used for different purposes. If this happens, it could be possible to have authentication or authorization bypasses. An attacker could take advantage of this to perform session puzzling attacks.

Java/J2EE	For example, different session keys must be used for different functionalities. An example of that is the “Forget username/password?” functionality that must have a different session key from that of the login one.
	
	Login functionality:
	`if(session.getAttribute("username") && credentialsOk())`
	`session.setAttribute("username",userName);`
	`else`
	`redirectToLoggedIn();`
	
	Forgotten password functionality:
	`if (userExists()){`
	`session.setAttribute("forgottenPW_Username",userName);`
	`..`
	`sendEmail(session.getAttribute("forgottenPW_Username"));`
	`}`
	
	The username value could be overwritten if used for both two functionalities and a malicious user firstly executes the second one.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Session_puzzling_(OTG-SESS-008)

4.3.27 IMPLEMENT A CORRECT LOGOUT FUNCTIONALITY

Requirement ID	SM-007
Priority	**Medium**
Description	To prevent an attacker from being able to reuse a previous session of a user who has not been able to log out, the logout function must always be present and reachable from every page of the application. It is necessary to make sure that the application invalidates the server-side session and deletes the client-side cookies. Make sure that the session is invalidated on the server side, for example through calls to the HttpSession.invalidate() function in Java, when the user logs out or when it remains inactive for a certain amount of time. Delete the cookies on the user's browser when the logout page is called up.

Java/J2EE	Example:
	`public void logout(HttpServletRequest request) throws ServletException {`
	`  HttpSession session = request.getSession(false);`
	`  if (session != null) {`
	`  session.invalidate();`
	`  request.logout(); // available only aServlet 3.0`
	`  }`
	`  loggedIn = false;`
	`  // Change current users role/state to not authenticated`
	`  ..`
	`  // Redirect to index page`
	`  ..`
	`}`

Java/Spring	The example below shows a logout made using the API provided by Spring Security. In this case, it is checked whether the user is authenticated and, if so, the session become invalidated, and the user redirected to the login page:
	`@RequestMapping(value="/logout", method = RequestMethod.GET)`
	`public String logoutPage (HttpServletRequest request, HttpServletResponse response) {`
	`  Authentication auth = SecurityContextHolder.getContext().getAuthentication();`
	`  if (auth != null){`
	`  new SecurityContextLogoutHandler().logout(request, response, auth);`

	} return "redirect:/login"; } By default, the logout() method will invalidate the session via session.invalidate(), we recommend to do not change this behavior.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_logout_functionality_(OTG-SESS-006)

4.3.28 PROTECT CLIENT/BROWSER AND SERVER COMMUNICATION

Requirement ID	CR-001
Priority	**Medium**
Description	Data such as session cookies, authorization codes, personal and sensitive data must be transmitted in encrypted sessions using the TLS protocol. Verify that the server and client negotiate the use of sufficiently robust ciphers and at the latest available version. Data protection must also be ensured within the system infrastructure in the exchange of components and application modules (eg Web Server, Application Server, DB server). If this control is not applied, it is possible for users who have access to the same subnet to carry out man In The Middle attacks. Refer to section 4.3.1 for examples of how to force communication over HTTPS. **References**: • https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet
Java/J2EE	Avoid using weak encoding such as Base64 to encode information within cookies. Communication channels between client-server must be encrypted using the TLS protocol.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Weak_SSL/TLS_Ciphers,_Insufficient_Transport_Layer_Protection_(OTG-CRYPST-001) • https://www.owasp.org/index.php/Testing_for_Sensitive_information_sent_via_unencrypted_channels_(OTG-CRYPST-003)

4.3.29 USE STANDARD CRYPTOGRAPHIC ALGORITHMS

Requirement ID	CR-002
Priority	**Medium**
Description	Applications must use libraries and cryptographic algorithms implementing state of the art security solutions that are as well recognized by scientific community for their robustness and security. Length of the encryption keys must be adequate to the type of the algorithm chosen. Private or sensitive data handled by the application, saved on a database or file, must be adequately protected by appropriate access permissions and technical and organizational measures. The encryption keys and the authentication certificates must be encrypted and maintained on a secure application server. Following the suggested standards. For the Asymmetric Cryptography (Public Key Encryption) use: • RSA with key length of 2048bit at minimum Symmetric Cryptography: • AES with key length of 256 bit at minimum Hashing algorithms: • SHA-256 or higher Password Hashing: • PBKDF2, Scrypt, Bcrypt In the case of creating hashes for storing passwords, add a salt (that is, a random sequence of bits) to the string on which the hash will be calculated (in fact it will be hash (password + salt)) to increase the level of complexity and make it safer against brute force attacks. The salt must be: • Long at least 32 or 68 bytes (256 or 512 bits); • Unique to every generation; • Generated using a cryptographically secure algorithm (as explained in paragraph 4.3.29); **References:** • https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet

<table>
<tr><td>Java/J2EE</td><td>The example below shows the use of PBKDF2 to generate password hash.

<pre>public static byte[] hashPassword(final char[] password, final byte[] salt, final int iterations, final int keyLength) {
  try {
    SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512");
    PBEKeySpec spec = new PBEKeySpec(password, salt, iterations, keyLength);
    SecretKey key = skf.generateSecret(spec);
    byte[] res = key.getEncoded( );
    return res;

  } catch(NoSuchAlgorithmException | InvalidKeySpecException e) {
    throw new RuntimeException(e);
  }
}</pre>
Salt must be a random value; the following code can be used to generate it:

<pre>public byte[] generateSalt() throws NoSuchAlgorithmException {
  SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
  byte[] salt = new byte[32];
  random.nextBytes(salt);
  return salt;
}</pre>
The iterations number must be at least 10,000 and the key length as 256.</td></tr>
<tr><td>Java/Spring</td><td>Spring Security provides different password hashing solutions. Specifically, it supports:

• BCrypt: BCryptPasswordEncoder class
• SCrypt: SCryptPasswordEncoder class
• Pbkdf2: Pbkdf2PasswordEncoder class (since version 4.1)

The use of the StandardPasswordEncoder class is not recommended because it uses SHA-256 for password hashing. Other not recommended encoders are the classics: Md5PasswordEncoder, Md4PasswordEncoder and ShaPasswordEncoder.

The example below shows how to configure XML to use the BCrypt as password encoder:

<pre><beans:bean name="bcryptEncoder"
        class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>

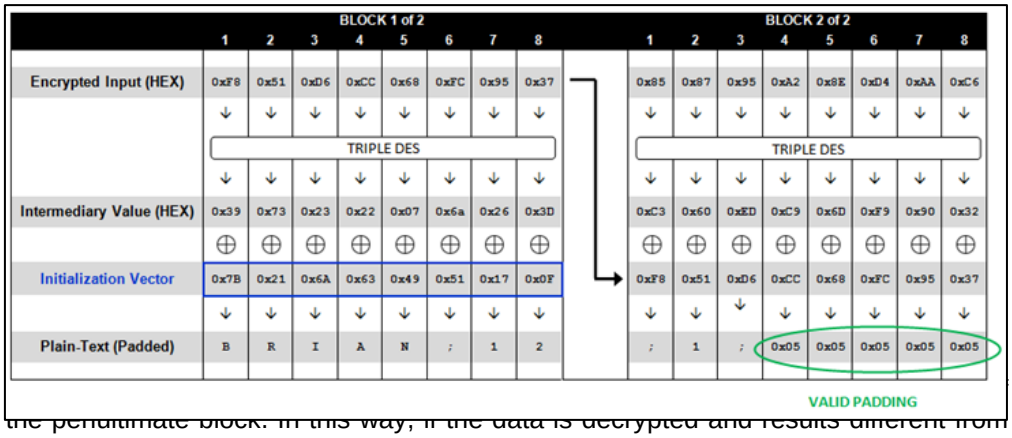
<authentication-manager></pre></td></tr>
</table>

	<authentication-provider>
	<password-encoder ref="bcryptEncoder"/>
	</authentication-provider>
	</authentication-manager>
	
	Similarly, in Java configuration:
	
	@Bean
	public PasswordEncoder passwordEncoder() {
	return new BCryptPasswordEncoder();
	}
	
	@Bean
	public DaoAuthenticationProvider authenticationProvider() {
	DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
	authenticationProvider.setPasswordEncoder(passwordEncoder());
	return authenticationProvider;
	}

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Weak_Encryption_(OTG-CRYPST-004) • https://www.owasp.org/index.php/Top_10-2017_A3-Sensitive_Data_Exposure • https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

4.3.30 PROTECT FROM PADDING ORACLE ATTACKS

Requirement ID	CR-003
Priority	**Medium**
Description	Padding Oracle attacks leverage the incorrect use of CBC (Chain Block Ciphers) algorithms. These algorithms are used to guarantee data confidentiality but result to be inadequate to ensure their integrity. According to PKCS#7 standard, a string encrypted by the CBC algorithm is decrypted correctly if the unused bytes of the last block have the number of the unused bytes as value. This convention is also note as padding and it is used to guarantee that a plaintext message can be subdivided in an exact number of blocks. The PKCS standard considers blocks of 8 or 16 bytes. During the decryption phase, each block is decrypted using a symmetric key algorithm. Then it is xored with the previous one. The following image illustrates the decryption process:

	 the penultimate block. In this way, if the data is decrypted and results different from 0x01 or the previous byte, the application will launch the Invalid Padding exception. By iterating through this process, it will be possible to obtain the initial clear text. The padding oracle attack allows decrypting any data without knowing the encryption key. In the same way, it is possible to encrypt any data. The following resources could be useful to gain more knowledge about the issue: • http://blog.mindedsecurity.com/2010/10/breaking-net-encryption-with-or-without.html • http://www.gdssecurity.com/l/b/2010/09/14/automated-padding-oracle-attacks-with-padbuster • http://netifera.com/research/
Java/J2EE	The suggested countermeasure against Padding Oracle attacks is known since many years: authenticated encryption. Authenticated encryption is a type of encryption that controls both authenticity and integrity at the same time. The most used mode is the one called Encrypt-then-MAC (EtM) in which the text is first encrypted, and the MAC code is generated starting from the ciphertext (which guarantees a good level of security); There are some symmetric encryption ways that natively support this feature and are documented in version 2.30 of the PKCS#11 standard. The suggested mode is the GCM one for the block ciphers and the CCM one for the stream ciphers (ex. CKM_AES_GCM and CKM_AES_CCM). These modes are supported by some of the most common security libraries such as BouncyCastle and Java version 7. **References**: • https://www.cryptsoft.com/pkcs11doc/STANDARD/pkcs-11v2-30-d1.pdf
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Padding_Oracle_(OTG-CRYPST-002)

4.3.31 SECURE RANDOM NUMBER IMPLEMENTATION

Requirement ID	CR-004
Priority	**Medium**
Description	The Pseudo Random Number Generators produce a bits sequence that are solely determined by an initial value called seed. The resulting sequence of a PRNG "seems" random, it is statistically analogous to random numbers sequences, but it is reproducible if the seed and the algorithm are known. A cryptographic PRNG has the additional property that the output is not predictable since the seed is not known. If it is required to generate tokens or sequences of non-predictable values, it is necessary to use specific cryptographic libraries.

Java/J2EE	It is strongly advised to not use the java.util.Random class which is proven to be predictable. Note that Math.random() is also not recommended because it uses java.util.Random.
	Instead, it is recommend using java.security.SecureRandom when cryptographically secure random numbers are needed.
	`SecureRandom random = new SecureRandom();`
	`byte bytes[] = new byte[20];`
	`random.nextBytes(bytes);`

Java/Spring	Spring Security provides the KeyGenerators class in the Crypto module, which allows generating random numbers using SecureRandom class.
Reference for verification	• https://www.owasp.org/index.php/Insecure_Randomness • https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

4.3.32 CORRECTLY MANAGE SECURITY EVENTS

Requirement ID	AL-001
Priority	**Medium**

Description	Depending on the application, it could be necessary to log the following events: • The start and the closing of the application; • The start and the end of the work session (authenticated users); • Actions performed by authorized users; • Invocation of processes or modules external to the application; • Controls on the data accessed by authenticated users; • Changes on the configurations of the application; • Addition/Extraction of removable devices; • Operations done on the database. Logs must exclude any sensible information, in other case must be encrypted and accessible only by the administrator.
Reference for verification	Manually inspect the code

4.3.33 PROTECT LOG FILES

Requirement ID	AL-002
Priority	**High**
Description	Log files must not be accessible, edited or deleted by non-authorized users. Unauthorized access to log files could lead to confidentiality or integrity issues resulting in data security issues.
Java/J2EE	Natively, Java offers the java.util.logging package that provides all the necessary classes to manage the logs. There are different external libraries that allow to manage the application logging, for example: • Log4J; • Commons logging; In order to protect log files from log forging problems, it is recommended to validate the input in order to remove those potentially harmful characters such as / n / r that would insert a new line in the logs. By inserting new lines in the log files, an attacker would be able to hide his actions by making it difficult to understand what happened. Refer to the paragraph 4.3.35 to see examples on how to validate data input. In addition to the input validation, it is also recommended to encode the string that will be inserted in the logs if these are to be read, for example, through a browser. Refer to the paragraph 4.3.36 for examples on how to output data encoding. **References**:

	• https://www.owasp.org/index.php/Log_Injection
Reference for verification	Manually inspect the code

4.3.34 PREVENT CACHING OF SENSITIVE DATA ON CLIENT SIDE

Requirement ID	DS-001
Priority	**Medium**
Description	Modern browsers, to help users who register to various systems, give the possibility to memorize passwords and pre-fill the forms with them. In addition, browsers also provide help in auto-completing forms when they contain fields that require data that was previously entered by the user. This data is saved on the hard disk and successively used to repopulate the form automatically. This mechanism is called autocomplete and, although it can be disabled by configuration, it is enabled by default on browsers and can represent a problem for the user's privacy depending on saved data. Sensitive data is then available to anyone with legitimate access to the machine - eg. shared browser in the office/family - or illegitimate - abusive, physical, or remote access -.

Java/J2EE	Since all browsers have the default autocomplete, the value must be turned off in the form in order to force the browser to not locally store what we consider sensitive data.
	The autocomplete attribute, standardized with HTML5, can be defined at the level of the whole form disabling the autocomplete for all input, as in the following example:
	`<form action="XXX" method="POST" autocomplete="off" >...</form>`
	The above code disables auto completion for all form data.
	If it is necessary to disable auto-completion for a limited set of inputs, it is possible to only explicit the attribute in the desired elements.
	Example:
	`<form action="XXX" method="POST" >`
	`...`
	`<input type="text" autocomplete="off" name="credit_card">`
	`...`

	</form> However, modern browsers will always ask the user to store the credentials related to a login form even if the developers have correctly set autocomplete to off, because is considered an advantage from the security point of view. For completeness it must be specified that the browsers give user the possibility to set a master password to access the saved passwords, but not by default. Despite this, to avoid caching even on login forms, it is still advisable to keep autocomplete set to off.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Vulnerable_Remember_Password_(OTG-AUTHN-005)

4.3.35 VALIDATE USER INPUT

Requirement ID	DV-001
Priority	**High**
Description	It is always advisable to prevent attacks as early as possible during request processing arriving from the user (attacker). Input validation can be used to identify unauthorized input before it is actually processed by the application. It is important to use a centralized routine system to validate application input data. Specify an adequate character set, like UTF-8, for all the input sources. Every input validation error must be rejected by the application. Validate the data after a redirect: a malicious user can present malicious contents circumventing the application logic and all the verifications executed before the redirect. Every input user data must always be validated before being used by the application. This control must be implemented on the data input flow at backend level that is on the application controller classes that regulate the data insertion and modification. It is recommended to follow the best practices reported below: • Controls and casting on the type; • Format controls; • Controls on characters subset (an email should allow a specific set of characters and not '<>'); • Control on the maximum length permitted; Check that the input has a value and a meaning equal to that requested by the application. Where possible, the best approach consists in using a whitelist of allowable values and reject the input that is not included inside them.

	If is not possible to use this approach, use regular expressions to make sure that only allowable characters can be inserted and that the input has a defined length. In every case, it is necessary to check at least the input type and range (setting a minimum and maximum length). It is also important to correctly sanitize: • Checks null byte (%00) • Checks for new line characters (%0d,%0a, \r, \n) Check for sequences such as "dot-dot-slash" (../ or ..\) alterations in the character path In cases where UTF-8 is supported as an extended character set, use an alternative representation such as: %c0%ae%c0%ae/. Furthermore, use canonicalization to protect against double encoding or other forms of obfuscation attacks. **References**: • http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet • https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

Java/J2EE	Use filters and validators depending on each input parameter as soon as it is used by the application.
	Example of input validation of username:
	`String name=request.getParameter("name");`
	`if(name!=null && !name.matches("^[:alpha:]{3,40}$")){`
	`throw SecurityException;`
	`}`
	
	If the expected input is a number, it should be casted to the native type and checked against the legitimate range.
	Example:
	`try{`
	`int id=Integer.parseInt(request.getParameter("id"));`
	
	`if(id<0 || id> 999999)`
	`throw SecurityException;`
	`}catch(NumberFormatException ex){`
	`throw SecurityException;`
	`}`

	Another approach is to use the validation beans that are part of the J2EE since version 6. By this way, the requirements to be respect are defined by annotations. If the data does not satisfy the requirements, the system will throw an exception.
	Example:
	`public class Utente {`
	`  @NotNull`
	`  @Size(min=3, max=40)`
	`  @Pattern(regexp="^[:alpha:]$")`
	`  private String firstname;`
	`  @NotNull`
	`  @Size(min=1, max=999999)`
	`  private Integer id;`
	`  ...`
	`}`
	To start the validation, the User instance must be annotated with @Valid when it is used (for example when passed to the controller as a parameter).
	`public @ResponseBody String createUser(@Valid Utente utente, BindingResult result, HttpServletResponse response){`
	`…`
	`}`

Java/Spring	Spring Framework supports bean validation since version 4.0, in order to validate user input, it is possible to use annotations as seen in the previous example. In addition to this approach, it is also possible to use the Validator interface which represents the classical method used in Spring validation. **References**: • https://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#validation-beanvalidation
Reference for verification	• https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet • https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting_(OTG-INPVAL-001) • https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_(OTG-INPVAL-002) • https://www.owasp.org/index.php/Testing_for_LDAP_Injection_(OTG-INPVAL-006) • https://www.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-008) • https://www.owasp.org/index.php/Testing_for_SSI_Injection_(OTG-INPVAL-009)

	• https://www.owasp.org/index.php/Testing_for_XPath_Injection_(OTG-INPVAL-010) • https://www.owasp.org/index.php/Testing_for_IMAP/SMTP_Injection_(OTG-INPVAL-011) • https://www.owasp.org/index.php/Testing_for_Code_Injection_(OTG-INPVAL-012) • https://www.owasp.org/index.php/Testing_for_Command_Injection_(OTG-INPVAL-013) • https://www.owasp.org/index.php/Testing_for_Buffer_Overflow_(OTG-INPVAL-014)

4.3.36 OUTPUT ENCODING

Requirement ID	DV-002
Priority	**High**
Description	It is important to perform data sanitization and data validation (that is the validation done on the application input data) before that data are shown to the user. By performing data encoding, it is possible to ensure that vulnerabilities like Stored XSS cannot interest the application. Attacks of this kind happen when the attacker is able to insert malicious code inside the database. Then this code is used to attack the user. It is important to use a standard routine, after having tested it, for each type of output encoding. Encode all characters although they are considered safe for the intended interpreter (SQL interpreter, XML, etc.) Sanitize all untrusted data output that is passed to a subsequent layer to be interpreted so that it is not considered a command. Some examples of layers normally used in an application are: • Operative System command; • SQL Query; • XML; • LDAP; **References**: • http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
Java/J2EE	When a string coming from not secure source must be inserted into HTML output, it should be encoded according to the Html context. Below there are some examples of context encoding where user input must be inserted. • TextNode and attribute that do not contains URL.

Change all non-alphanumeric values to HtmlEntities (from character to &#xXX;):

```
public String encode(String input) {
  StringBuffer sb = new StringBuffer();
  for (int i=0; i<input.length(); i++) {
          char c = input.charAt(i);
          sb.append(encodeCharacter(new Character(c)));
  }
  return sb.toString();
}
public String encodeCharacter(Character c) {
  return "&#" + (int)c.charValue() + ";";
}
```

This function can then be used when the input is included in the response page:

```
<div><%=encode(request.getParameter("name"))%></div>
```

Otherwise:

```
<input type="text" name="id" value="<%=encode(request.getParameter("name"))%>">
```

- Link and similar (from characters to %XX):

Change all values in URL Encoding (java.net.URLEncoder.encode()):

```
<a
href="http://Host/?par=<%java.net.URLEncoder.encode(request.getParameter("param"
), "UTF-8")%>">click</a>
```

It is advisable not to allow the user to control the whole URL of the link, as done in the previous example, in order to not lead to other security problem.

- JavaScript Context

Create an encoding method and use it for JavaScript strings (from character to \xXX):

```
public String encode(String input) {
  StringBuffer sb = new StringBuffer();
  for (int i=0; i<input.length(); i++) {
          char c = input.charAt(i);
          sb.append( encodeCharacter(new Character(c)));
  }
  return sb.toString();
}
```

```
public String encodeCharacter(Character c) {
                    char ch = c.charValue();
                    if (ch == 0x00) return "\\0";
                    if (ch == 0x08) return "\\b";
                    if (ch == 0x09) return "\\t";
                    if (ch == 0x0a) return "\\n";
                    if (ch == 0x0b) return "\\v";
                    if (ch == 0x0c) return "\\f";
                    if (ch == 0x0d) return "\\r";
                    if (ch == 0x22) return "\\\"";
                    if (ch == 0x27) return "\\'";
                    if (ch == 0x5c) return "\\\\";
          ...
                    // encode up to 256 with \\xHH
       String temp = Integer.toHexString((int)ch);
          if (ch <= 256) {
                String pad = "00".substring(temp.length());
                return "\\x" + pad + temp.toUpperCase();
          }

          // otherwise encode with \\uHHHH
       String pad = "0000".substring(temp.length());
       return "\\u" + pad + temp.toUpperCase();
}
```

This function will then be used when user input is added in the JavaScript context:

```
<script> f="value<%=encode(request.getParameter("value"))%>"
</script>
```

- Attributes that identify Css styles or Css files:

Create an encoding method and use it for css strings (from characters to \\XX):

```
 public String encode(String input) {
       StringBuffer sb = new StringBuffer();
       for (int i = 0; i < input.length(); i++) {
          char c = input.charAt(i);
          sb.append(encodeCharacter(new Character(c)));
       }
       return sb.toString();
  }
 public String encodeCharacter(Character c) {
       char ch = c.charValue();

       // encode up to 256 \x syntax
       if (ch <= 256) {
          return "\\" + ch;
       }
```

```
        // otherwise encode with \\HHHHHH
        String temp = Integer.toHexString((int) ch);
        return "\\" + temp.toUpperCase() + " ";
    }
```

Note 1: JavaScript and CSS contexts refer only to strings. The encoding, however, loses its meaning if the concatenation with untrusted data takes place outside a JavaScript / Css string.

Note 2: encoding must be done when the data are concatenated with the Html. The encoded data must not be subsequently manipulated or could lose the encoding benefits.

Note 3: If you are dealing with XML, the encoding must be done by executing the transformation specifications to XMLEntities.

Finally, it should be considered that the JSP pages can use the Java Standard Tag Library (JSTL) library that exposes the <c: out> tag through which it is possible to encode the output in a secure way. This library is also commonly used in Spring, so more is discussed in the next paragraph.

Java/Spring

Spring provides different solutions to show the output securely within the page.

Java Standard Tag Library

A common way is to use the <c:out> tags, offered by JSTL, within the jsp pages. This tag performs by default the encoding of potentially harmful characters (for example: <,>, ", '). We therefore recommend using this tag when information needs to be presented on the page and the source is considered not secure.

- TextNode: use <c:out>:

```
<p>
 Hello <b><c:out value="${request.remoteUser}"/></b>
</p>
```

- Link: use <c:url > and <c:param>:

```
<c:url value="/myAction.do" var="url">
<c:param name="param1" value="${user.fullName}"/>
<c:param name="param2" value="${request.param}"/>
</c:url> <a href="${url}">Click</a>
```

Spring Tag Library

In addition to the JSTL, Spring provides some custom tags that facilitate data binding and can be used together with the JSTL in the jspc. All the tags available in Spring have an attribute to enable or disable the HTML escaping, which is false by default. Spring also

	provides the form tag library which offers tags that facilitate the management of forms and, unlike generic tags, have the HTML escaping set to true by default.
	It is therefore recommend the use of form library tags for rendering external input.
	If the usage of generic tags is needed, to make sure that the HTML characters are escaped, put a configuration as shown below in the web.xml that enables the escaping for all the tags at the application level.
	`<context-param>`
	`<param-name>defaultHtmlEscape</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`
	It is also possible to enable escaping it at the specific page level through the following tag (add it at the top of the page):
	`<spring:htmlEscape defaultHtmlEscape="true" />`
	Otherwise, the latest option is to enable HTML escaping for the single tag by acting on its attribute.
	Thymeleaf
	Thymeleaf is a Java template engine that integrates perfectly with Spring and introduces some attributes populated at runtime with the values taken from the model. This engine offers by default the escaping in HTML context.
	Below there is an example of Thymeleaf usage, where if the user is authenticated, a button is presented to logout and the user's name is written on the page. If the user's name contained characters that were not allowed, they would be encoded:
	`<div th:if="${#httpServletRequest.remoteUser != null}">`
	`<form class="navbar-form pull-right" th:action="@{/logout}" method="post">`
	`<input type="submit" value="Log out" />`
	`</form>`
	`<p class="navbar-text pull-right" th:text="${#httpServletRequest.remoteUser}">`
	`sample_user`
	`</p>`
	`</div>`
	Programmatic approach
	If it's needed to encode in a controller class instead of view page, you can use the org.springframework.web.util.HtmlUtils.htmlEscape()method.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_(OTG-INPVAL-002)

	• https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting_(OTG-INPVAL-001)

4.3.37 USE OF PREPARED STATEMENTS

Requirement ID	DV-003
Priority	**High**
Description	The application is affected by SQL Injection vulnerabilities when it uses non-validated user's input to make queries on the database. It is mandatory to not use strings concatenation with input data to create the queries, but always use prepared statements when it is necessary to make queries using input data provided by the user. This allows to have an automatic escape of all the characters that are potentially harmful in SQL. In this way, the SQL command logic is separated from the passed data. The user through which the application makes database operations must have the lowest privileges possible and access only to the information strictly needed. **In-depth analysis**: • http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Java/J2EE	Use, where the DBMS makes it available, the prepared statements.
	`String userId = request.getParameter("userId");`
	`String query = "SELECT * FROM User WHERE userId = ?";`
	`PreparedStatement prepStmt = connection.prepareStatement(query);`
	`prepStmt.setString(1, userId);`
	`ResultSet rs = prepStmt.executeQuery();`
	
	Prepared statements separate data from the SQL query, making it impossible to inject SQL commands.
	Of course, untrusted values must not be concatenated in the template query, otherwise SQL Injection problems will still be present.

Java/Spring	Moreover, Spring Framework provides the jdbcTemplate class which use prepared statements by default, avoiding SQL injection problem.
	Example:
	`int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(`
	`"select count(*) from t_actor where first_name = ?", Integer.class, "Joe");`

Reference for verification	• https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Escaping_Dynamic_Queries • https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)

4.3.38 CORRECTLY BUILD HTTP REQUESTS

Requirement ID	DV-004
Priority	**High**

Description	When a value like a URL has to be validated, it must be parsed and then check all parts which will be used from application.
	The URLs related to the HTTP protocol are like this:
	protocol://username:password@host.name.ltd[:port]/pathName[pathInfo]/?queryString#URLFragment
	
	URL parsing is not a trivial task, and it is strongly related to the charset and to how the layer handles specific values.
	At this time, the object instantiated by the URL parsing can be described by the following attributes:
	• origin: protocol, hostname and port (ex. https://www.example.com:8080);
	• protocol: the protocol;
	• username: the username as described before;
	• password: the password as described before;
	• host: the host consists in hostname and port;
	  ○ hostname: the hostname as described before;
	  ○ port: the port as described before;
	• pathname: the pathname as described before;
	• search: the query string including the character “?”
	• hash: the URL fragment including the character “#”
	
	The developer has to care about the validation and the eventual additional parsing of the URL single components that are needed by the source code.
	In addition, it is important to make particular attention when the protocol is checked.

	Due to the format complexity, it is strongly suggested to use a semantic whitelist approach.
	To prevent Server-Side Request Forgery attacks, if part of the new URL can be tampered from external inputs, perform an extensive validation of parsed URL elements.
	The following example shows an application that is requesting a URL from user supplied inputs to perform an internal request:
	`url=HTTP://XXX/PATH?QUERYSTRING`
	The URL in the URL parameter should be parsed by using a trustworthy URL parser and validated in any single element:
	`proto=HTTP&host=XXX&path=PATH&query=QUERYSTRING`
	The same could be valid during the construction of POST requests.
	Any untrusted input must be validated and escaped in relation to the context where will be used.

Java/J2EE	It is suggested to use custom URL parsing algorithms that are strongly related to the used libraries and frameworks. This can prevent attacks like “Server-Side Request Forgery”. Please refer to paragraph 4.3.35 for examples on how to validate the individual elements that make up the URL.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_HTTP_Parameter_pollution_(OTG-INPVAL-004)

4.3.39 CORRECTLY HANDLE APPLICATION ERRORS

Requirement ID	EH-001
Priority	**Medium**
Description	The application must be able to report to the users all the errors or exceptions through appropriate and comprehensible messages. These must not disclose information that could be used for later attacks. Error messages must not include: • Variable names and types; • SQL strings; • Source code parts; • Web server and DB server names and versions; • Exceptions;

	• Application information; Error messages must be handled by a generic courtesy page. Verify that the application does not send confidential information, such as testing code within the Java Applet or HTML. Remove all the unnecessary comments and verify that the application does not send private information via errors.

Java/J2EE	Edit web.xml file as shown below:
	
	`<error-page>`
	`  <exception-type>java.lang.Exception</exception-type>`
	`  <location>/server_error.jsp</location>`
	`</error-page>`
	`<error-page>`
	`  <error-code>500</error-code>`
	`  <location>/server_error.jsp</location>`
	`</error-page>`
	`<error-page>`
	`  <error-code>404</error-code>`
	`  <location>/file_not_found.jsp</location>`
	`</error-page>`
	
	server_error.jsp and file_*not_found.jsp* could contain the code to log the request that generated the error.
	
	Finally, it should be considered that the error shown to the user must be as generic as possible, carrying out a session cleaning and a redirect to the index page.

Java/Spring	The general considerations expressed above are valid for this framework, so the error messages shown must not be contain detailed information on the application and the technologies in use (such as stack-traces), but must be as generic as possible while the details must be logged.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_Error_Code_(OTG-ERR-001) • https://www.owasp.org/index.php/Testing_for_Stack_Traces_(OTG-ERR-002)

4.3.40 USE THE X-FRAME-OPTIONS HEADER

Requirement ID	SH-001
Priority	**Low**

Description	To avoid attacks related to *UI-Redressing* and based on transparent iframes, it is suggested to add server side, in the all the responses that involve text/HTML content, the following header:
	`X-Frame-Options: SAMEORIGIN`
	The previous value is recommended if the application uses iframes that contain pages from the same source, otherwise if the application does not need to use iframes use the following one:
	`X-Frame-Options: DENY`
	References:
	• https://developer.mozilla.org/en-US/docs/Web/HTTP/X-Frame-Options

Java/J2EE	Below there is an example of how to set the header programmatically within a filter that is called *ClickjackPreventionFilter*:
	`public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {`
	`  HttpServletResponse res = (HttpServletResponse)response;`
	`  res.setHeader("X-Frame-Options", "SAMEORIGIN");`
	`  chain.doFilter(request, response);`
	`}`
	This filter should be added to the web.xml to cover all pages:
	`<filter>`
	`  <filter-name>ClickjackPreventionFilter</filter-name>`
	`  <filter-class>com.org.ClickjackingPreventionFilter</filter-class>`
	`</filter>`
	`<filter-mapping>`
	`  <filter-name>ClickjackPreventionFilter</filter-name>`
	`  <url-pattern>/*</url-pattern>`
	`</filter-mapping>`

Java/Spring	Spring Security adds by default some headers to help making the application more secure. The following header is added by default:

	X-Frame-Options: DENY The following example shows how to configure the aforementioned header: <http> <!-- ... --> <headers> <frame-options policy="SAMEORIGIN" /> </headers> </http>
Reference for verification	• https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet#Defending_with_X-Frame-Options_Response_Headers • https://www.owasp.org/index.php/Testing_for_Clickjacking_(OTG-CLIENT-009) • https://www.owasp.org/index.php/OWASP_Secure_Headers_Project

4.3.41 USE THE X-XSS-PROTECTION HEADER

Requirement ID	SH-002
Priority	**Low**
Description	The X-XSS-Protection header enables XSS filters implemented in the latest generation of Chrome, Safari and Internet Explorer 8+ (Firefox has an extension called NoScript that takes care of this task). Although this header has somehow been superseded by the Content-Security-Policy header, it is recommended to use it to provide protection against older user agents that do not support CSP. The following example enables the anti xss filter and prevents the page from appearing if a problem is detected: X-XSS-Protection: 1; mode=block **Limitations**: • Each response must contain the header with the directive since it is a countermeasure per-page; • Anti XSS filters must be generic and should be considered as an additional feature in order to help and not as a substitute of output encoding; • DOM Based XSS are not always identified by these filters.

	References: • http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx • http://msdn.microsoft.com/en-us/library/dd565647%28v=vs.85%29.aspx

Java/J2EE	Below there is an example of how to set up the header inside a filter:
	`public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {`
	`  HttpServletResponse res = (HttpServletResponse)response;`
	`  res.setHeader("X-XSS-Protection", "1; mode=block");`
	`  chain.doFilter(request, response);`
	`}`
	
	This filter should be added to the web.xml to cover all pages:
	`<filter>`
	`  <filter-name>XSSProtectionFilter</filter-name>`
	`  <filter-class>com.org.XSSProtectionFilter</filter-class>`
	`</filter>`
	
	`<filter-mapping>`
	`  <filter-name>XSSProtectionFilter</filter-name>`
	`  <url-pattern>/*</url-pattern>`
	`</filter-mapping>`

Java/Spring	Spring Security adds by default some headers to help making the application more secure.
	The following header is added by default:
	`X-XSS-Protection: 1; mode=block`
	
	The following example shows how to configure the aforementioned header:
	`<http>`
	`  <!-- ... -->`
	`  <headers>`
	`  <xss-protection block="false"/>`
	`  </headers>`
	`</http>`

Reference for verification	• https://www.owasp.org/index.php/OWASP_Secure_Headers_Project

4.3.42 USE HTTP STRICT TRANSPORT SECURITY

Requirement ID	SH-003
Priority	**Low**
Description	The HTTP Strict-Transport-Security (HSTS) forces the browser to exclusively do secure connections (HTTPS) to the server that implements it. This reduces the impacts related to Man in the Middle attacks. An HSTS header has two directives: • max-age: indicates the number of seconds for which the browser must consider the header valid and then convert any HTTP request into an HTTPS request to the site. • includeSubDomains: indicates that subdomains must also be addressed using HTTPS. If possible, it is recommended using this header to force the browser to communicate only via HTTPS. E.g.: `Strict-Transport-Security: max-age=16070400; includeSubDomains`

Java/J2EE	The example below shows how to set the header in a filter:
	`public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {`
	`  HttpServletResponse res = (HttpServletResponse)response;`
	`  res.setHeader("Strict-Transport-Security", "max-age=16070400; includeSubDomains");`
	`  chain.doFilter(request, response);`
	`}`
	This filter should be added to the web.xml to cover all the pages:

	<filter> <filter-name>HSTSFilter</filter-name> <filter-class>com.org.HSTSFilter</filter-class> </filter> <filter-mapping> <filter-name>HSTSFilter</filter-name> <url-pattern>/*</url-pattern> </filter-mapping>
Java/Spring	Spring Security adds by default some headers that help to improve the application security. The following header is added by default: Strict-Transport-Security: max-age=31536000 ; includeSubDomains The following example shows how to configure the aforementioned header: <http> <!-- ... --> <headers> <hsts include-subdomains="true" max-age-seconds="31536000" /> </headers> </http> Please note that this header is added by Spring only if the application is using a secure connection over HTTPS.
Reference for verification	• https://www.owasp.org/index.php/Test_HTTP_Strict_Transport_Security_(OTG-CONFIG-007)

4.3.43 USE THE CONTENT-SECURITY-POLICY HEADER

Requirement ID	SH-004
Priority	**Low**

Description	The use of the Content-Security-Policy header is strongly suggested because it instruct the browser to restrictively allow behaviors using a whitelist approach. Because the CSP has an important impact on the operation of the pages, the directives that have to be sent to the browser need a fine tuning. For example, it is possible to completely block all the JavaScript inlines and the evals, and all the HTML attribute events. CSP can prevent different categories of attack, such as Cross-Site-Scripting, UI Redressing, malicious JavaScript code and others. **Limitations**: • Each response must contain the header with the directive since it is a countermeasure per-page; **References**: • https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP • http://www.cspplayground.com/resources

Java/J2EE	The CSP is allowed at the <*Meta*> level header even if not all the browsers support it.
	In general, a CSP header is defined as follow:
	`Content-Security-Policy: policy`
	The policy is created through some directives usage. The following example show how to give users the ability to include images from any source while restricting access to media and scripts:
	`Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com media2.com; script-src userscripts.example.com`
	The example below shows how to programmatically set the header inside a filter:
	`public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {`
	`  HttpServletResponse res = (HttpServletResponse)response;`
	`  res.setHeader("Content-Security-Policy", "default-src 'self'; img-src *; media-src media1.com media2.com; script-src userscripts.example.com");`
	`  chain.doFilter(request, response);`
	`}`

	This filter should be added to the web.xml to cover all pages:
	`<filter>`
	`  <filter-name>CSPHeaderFilter</filter-name>`
	`  <filter-class>com.org.CSPHeaderFilter</filter-class>`
	`</filter>`
	
	`<filter-mapping>`
	`  <filter-name>CSPHeaderFilter</filter-name>`
	`  <url-pattern>/*</url-pattern>`
	`</filter-mapping>`

Java/Spring	Spring Security allow to add the header through a configuration as shown in the following example (or via Java configuration):
	`<http>`
	`  <!-- ... -->`
	`  <headers>`
	`  <content-security-policy`
	`  policy-directives="script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report-endpoint/" />`
	`  </headers>`
	`</http>`

Reference for verification	• https://www.owasp.org/index.php/OWASP_Secure_Headers_Project • https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet#Defending_with_Content_Security_Policy_frame-ancestors_directive

4.3.44 PROTECT THE ADMINISTRATIVE INTERFACES

Requirement ID	CM-001
Priority	**Medium**
Description	It is necessary to not expose application management consoles or, at least, allow their access only to local users (Intranet). The access can be implemented internally via an SSH tunnel or externally through a VPN. Make sure that default credentials are not used, and robust password policies are implemented. This requirement must also be applied to all Jboss management consoles where used.

Java/J2EE	Edit the web.xml file of the administration application with the following values (the same configuration can be done via annotations):
	
	`<security-constraint>`
	`  <web-resource-collection>`
	`  <url-pattern>/*</url-pattern>`
	`  </web-resource-collection>`
	`  <auth-constraint>`
	`  <role-name>admin</role-name>`
	`  </auth-constraint>`
	`</security-constraint>`
	
	It is also recommended to not insert any specific http-method in the access restriction, since it could lead HTTP method vulnerabilities (HTTP Verb Tampering).

Reference for verification	• https://www.owasp.org/index.php/Enumerate_Infrastructure_and_Application_Admin_Interfaces_(OTG-CONFIG-005)

4.3.45 DISABLE DIRECTORY LISTING

Requirement ID	CM-002
Priority	**High**
Description	The directory listing allows site visitors to view folders content, simply by accessing the URL of directory via browser if there is no index page or default page. To avoid this, it is suggested to make sure that the server is correctly configured with the directory listing disabled.

Java/J2EE	Disabling the listing directory may depend on the server used.
	
	For example, on application servers such as Tomcat by editing the web.xml file (in the most recent versions it should be disabled by default):
	
	`<init-param>`
	`  <param-name>listings</param-name>`
	`  <param-value>false</param-value>`
	`</init-param>`

Reference for verification	• https://www.owasp.org/index.php/Test_Application_Platform_Configuration_(OTG-CONFIG-002)

4.3.46 DISABLE DANGEROUS HTTP METHODS

Requirement ID	CM-003
Priority	**Low**

Description	Disable HTTP methods: TRACE, TRACK, PUT, DELETE;
	On Apache servers, add the following line for each virtual host in the configuration file to disable TRACE and TRACK:
	`RewriteEngine on`
	`RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)`
	`RewriteRule .* - [F]`

Java/J2EE	The following configuration will not make the HTTP TRACE, DELETE, PUT, TRACK methods available (the same configuration can be done via annotations):
	`<security-constraint>`
	`<web-resource-collection>`
	`<web-resource-name>restricted methods</web-resource-name>`
	`<url-pattern>/*</url-pattern>`
	`<http-method>TRACE</http-method>`
	`<http-method>DELETE</http-method>`
	`<http-method>TRACK</http-method>`
	`<http-method>PUT</http-method>`
	`</web-resource-collection>`
	`<auth-constraint />`
	`</security-constraint>`

Reference for verification	• https://www.owasp.org/index.php/Test_HTTP_Methods_(OTG-CONFIG-006)

4.3.47 REMOVE SYSTEM DEFAULT ACCOUNTS

Requirement ID	CM-004
Priority	**Medium**
Description	The application platforms, the databases, and the entire environment on which the application runs, could be attacked. Make sure that all the platforms do not generate default and easily guessable accounts.

	It is also recommended to make sure that the infrastructure on which the application must be installed does not have any active default account. If there are any default accounts, it is advised to delete them or, if necessary, set a strong password.
Reference for verification	• https://www.owasp.org/index.php/Testing_for_default_credentials_(OTG-AUTHN-002)

4.3.48 REMOVE UNUSED FILES

Requirement ID	CM-005
Priority	**Medium**
Description	During the software development it happens that the developer creates test or temporary files that are needed to make tests or for backup. When the software is in production, it is necessary to make sure that such files are removed, and that files with the following extensions: • .old, • .bak • Zip archive and similar are not present if they are not really needed. These files must be protected from unauthorized access since they usually contain private or sensitive data.
Reference for verification	• https://www.owasp.org/index.php/Review_Old,_Backup_and_Unreferenced_Files_for_Sensitive_Information_(OTG-CONFIG-004)

4.3.49 PROTECT FROM HTTP VERB TAMPERING

Requirement ID	CM-006
Priority	**Medium**
Description	The *HTTP Verb Tampering* is an attack that leverages the use of HTTP methods supported by the server to bypass the authentication and authorization mechanisms implemented by the target application. This could be possible in case the application handles the access only for specific HTTP methods, so allowing the access through the use of the other ones. It is possible to address this problem following these steps:

	• Configure the "deny all" for all methods that are not strictly necessary for the application operation; • Ensure that access to resources is controlled indifferently by the method used; • Disable, if not strictly necessary, the HEAD method; • Do not accept direct requests to JSP files, but use a controller that uses the RequestDispatcher.forward() method to send a request to a JSP file;

Java/J2EE	The following example show how to properly configure the application to block the resources access (the same configuration can be done via annotations).
	`<security-constraint>`
	`  <display-name>Example Security Constraint</display-name>`
	`  <web-resource-collection>`
	`  <web-resource-name>Protected Area</web-resource-name>`
	`  <url-pattern>/security/protected/*</url-pattern>`
	`  </web-resource-collection>`
	`</security-constraint>`
	It is not correct, to specify methods using the <http-method> tag in the configuration. The lack of this tag in the security constraint implies that the check will be carried out regardless of the method used, thus preventing HTTP verb tampering issues.

Reference for verification	• https://www.owasp.org/index.php/Testing_for_HTTP_Verb_Tampering_(OTG-INPVAL-003)

4.3.50 AVOID USE OF PRIVATE EMBEDDED DATA IN HTML CODE

Requirement ID	CM-007
Priority	**Medium**
Description	Anything that the user visualizes on his browser can be read by the user himself by clicking on “show page HTML source”. The application should not leak any private information inside the HTML source of the page. In particular, consider: • Test code (commented or not); • Test data (used or not); • Comments containing debug data or other private information;

	E.g.: • Username and password precompiled in a form to access specific layer • <!-- SELECT * from TableName where col1=XX ... --> • /* Test User: testUser Pass: testPassword */ Although blurred in some way, all data coming to the browser must be considered as readable.
Reference for verification	• https://www.owasp.org/index.php/Review_webpage_comments_and_metadata_for_information_leakage_(OTG-INFO-005)

4.3.51 CORRECTLY CONFIGURE EXTENSIONS HANDLING

Requirement ID	CM-008
Priority	**Medium**
Description	It is suggested to configure the web server in order to do not give read access in case file requested it's used by the server itself but in some way reachable from the web. It is recommended to pay particular attention to files requested by the user that have an extension that is not normally managed by the web server.

Java/J2EE	Below there is an example of common extension blocking access made by editing the web.xml file (the same configuration can be done via annotations)
	
	`<security-constraint>`
	`  <web-resource-collection>`
	`  <url-pattern>*.asa</url-pattern>`
	`  <url-pattern>*.bak</url-pattern>`
	`  <url-pattern>*.class</url-pattern>`
	`  <url-pattern>*.inc</url-pattern>`
	`  <url-pattern>*.jar</url-pattern>`
	`  <url-pattern>*.log</url-pattern>`
	`  <url-pattern>*.properties</url-pattern>`
	`  <url-pattern>*.sql</url-pattern>`
	`  <url-pattern>*.zip</url-pattern>`
	`  </web-resource-collection>`
	`  <auth-constraint />`
	`</security-constraint>`

Reference for verification	• https://www.owasp.org/index.php/Test_File_Extensions_Handling_for_Sensitive_Information_(OTG-CONFIG-003)

4.3.52 USE COMPONENTS WITHOUT KNOW VULNERABILITIES

Requirement ID	CM-010
Priority	**Medium**
Description	All the software, such as: • libraries used for a specific functionality • the framework used to develop • the DBMS • the operating system should be considered safe until a vulnerability is found. It is strongly suggested, when installing new libraries, to check for known vulnerabilities and if possible find the version that fixes the vulnerability itself. This way, vulnerabilities are not introduced due to lack of process.
Implementation	It is suggested the adoption of the following rules related to the installation and maintenance of third-party software: • Remove unused dependencies, components, files and documentation. • Use version control and vulnerability presence tools such as Dependency Check, Retire.js etc. • Use official repositories and if possible internal repositories to the enterprise to avoid running into backdoors. • Download the software on a secure channel. • Prefer signed packages • Where it will not be possible to update a component with vulnerabilities consider the use of the so-called virtual patching, or configure the IPS / IDS system to detect attacks on the known vulnerability.
Reference for verification	• https://www.owasp.org/index.php/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities

5 CHECKLIST FOR REQUIREMENT ACCEPTANCE

For the acceptance of the requirements, the following criteria are required:

- Have the requirements been clearly explained in chapter 3?

- Have the requirements been numbered and prioritized?
- Does each requirement meet these characteristics?
    - Complete: necessary information must not be left out.
    - Correct: each requirement must accurately describe the functionality to be implemented.
    - Feasible: it must be possible to implement the requirement with the known possibilities and limitations of the system and the environment.
    - Necessary: the requirement must document something that is actually needed by the customer or by an external requirement, an external interface, or a standard.
    - Prioritized: a Priority must be assigned to the requirement in order to indicate the importance of including it in a specific release of the product.
    - Unambiguous: the requirement must be written in a concise, simple manner, in the language of the user's domain, so that anyone who reads the requirement can give a single interpretation and different readers reach the same conclusion.
    - Verifiable: It must be possible to carry out tests for the requirement in order to verify correct implementation.

5.1 CHECKLIST

The following table summarizes the set of security requirements to be implemented in the development of secure software for J2EE/Spring web applications:

Category	Check to implement	Requirement Implemented
Authentication	Send private information over encrypted channels	
	Protect from User Enumeration	
	Protection from guessable (dictionary) user account	
	Implement a strong password policy	
	Avoid authentication bypass for private resources	
	Implement a correct password reset method	
	Avoid caching sensitive data	
	Avoid positive authentication	
	Remove application default accounts	
	Verify wrong authentication attempts	
	Implement a secure password change functionality	
	Send private information via POST	

Category	Check to implement	Requirement Implemented
Access Control	Protect from Path Traversal	
	Protect resources from unauthorized access	
	Avoid privilege escalation	
	Avoid authorization bypass	
	Correctly manage third party code	
	Correctly handle Cross Origin (CORS) resources	
	Implementing anti automation controls	
	Correct use of Websockets	
Session Handling	Build a secure HTTP session	
	Set secure attributes for the session cookies	
	Renew the HTTP session after successful login	
	Protect from Cross Site Request Forgery	
	Check the uniqueness of the user's session	
	Isolate session keys	
	Implement a correct logout functionality	
Data Protection - Cryptography	Protect client/browser and server communication	
	Use standard cryptographic algorithms	
	Protect from Padding Oracle attacks	
	Secure Random Number implementation	
Audit e Logging	Correctly manage security events	
	Protect log files	

Category	Check to implement	Requirement Implemented
Sensitive Data Handling	Prevent caching of sensitive data on client side	
Data Validation	Validate user input	
	Output encoding	
	Use of prepared statements	
	Correctly build HTTP requests	
Error Handling	Correctly handle application errors	
HTTP Headers	Use the X-Frame-Options header	
	Use the X-XSS-Protection header	
	Use HTTP Strict Transport Security	
	Use the Content-Security-Policy header	
Configuration Management	Protect the administrative interfaces	
	Disable directory listing	
	Disable dangerous HTTP methods	
	Remove system default accounts	
	Remove unused files	
	Protect from HTTP Verb Tampering	
	Avoid use of private embedded data in HTML code	
	Correctly configure extensions handling	
	Use components without know vulnerabilities	