TSS-WEB

SECURITY STANDARD FOR WEB DEVELOPMENT

Template

A Standard by Secodis GmbH

Version 2.0
1 November 2020

About TSS-WEB

This document consists of best practices that can be used in a security standard for web-based applications and services. It may be used as a template for a custom organization-specific security standard or just a collection of suggestions of baseline requirements for teams and projects. All requirements in this document are based on common best practices in combination with our own experiences in this field.

The latest version of this document is available at https://tss-web.secodis.com in various file formats and languages. You'll also find a way to contact the authors of this document on this site that you can use if you have any questions, suggestions or feedback.

License

Matthias Rohr
Secodis GmbH

Document History

Version	Date	Changes
1.0	20/02/2015	Initial release (German)
…	…	…
1.3 (Draft)	29/08/2016	Initial English version based on a complete review of the standards with a lot of improvements and corrections.
1.3.1 (Draft)	28/11/2016	– Changes to CSP statements – CVSS scoring added to the security of 3rd party components
1.4 (Draft)	31/03/2016	– New requirements for handling X.509 certificates (8.17) – Changes for agile Development (chapter 5) – “Use of Secure JavaScript APIs” moved from 8.4 (“Output Validation”) to 8.14 (“Client-Side Security”) – Various changes to chapter 6 ("Security Tests) – Security Auditor renamed in Security Officer
1.5	15/08/2017	– Security officer renamed to IT security function – OWASP Top Ten was removed and will be maintained on the web site (tss-web.secodis.com) in the future. – Protection class has been renamed into assurance class – Multiple modifications of chapter 5 (“security in development process”) – Multiple modifications of chapter 7 (“supplier requirements”) – Minor modifications to file upload requirements – Requirements for HPKP removed and for referrer policy added
1.6	19/03/2019	– Section 1.3 (“Roles”): Changes of role description “Security Champion” and role “developer” added – Chapter 3 (“Operational Requirements”): Extension of DMZ restrictions and security monitoring – Multiple modifications of chapter 5 (“Protection of Source and Program Code”) – 8.2 (“Input Validation”): New requirement for secure object sterilization – 8.5 (“Authentication & Registration of Users): Deprecated NIST standard replaced with current NIST SP 800-63B – 8.8 (“Authentication at Backend”): Multiple modifications – 8.11 (“Access Controls”) Modification of OAuth requirements. Move of CORS and OAuth requirements to Service Security – 8.13 (“Management of Technical Keys & Secrets”) Multiple Modifications – 8.16 (“XML Parser Security”) Multiple modifications, renaming to “Service Security”; XML parser security requirements moved to 8.2 (“Input Validation”).

		– Appendix A: Integration von SameSite cookies and modification of CSP requirements
1.7	24/05/2019	– Role “Security Champion” extended – 3rd party component renamed to 3rd party dependencies and relevant requirements in chapter 4 completely revised. – Chapter 5 (“Security of Development Process”) revised – Chapter 6 (“Security Tests”) revised, and test policy aligned to assurance classes – Update of security logging and monitoring requirements
1.8	01/07/2019	– Chapter 3 (“Operational Requirements”) revised and made more compliant to container and cloud environments. – Chapter 5 (“Security of Development Process”) completely revised and restructured – Chapter 6 (“Security Tests”) completely revised and and restructured – Name of the standard changed to “web development” standard instead of “web application” standard. – Minor updated in other chapters.
2.0	01/11/2020	Complete revision, including chapters: – Chapter 3 („Secure Operation“) – Chapter 5 („Secure Development Process“) – Chapter 6 („Security Tests“) – Section 8.11. („Data Security & Cryptography“) – Section 8.14 („Service & API Security”)

Security Standard for Web Development

Example Inc.

Version: 1.0
Classification: INTERNAL

Table of Contents

Document Properties

Location	**tbd**
Owner	**tbd**
Type	**Technical Standard**
Classification	**For internal use only (INTERNAL)**
Next Review	**tbd**

Referenced Documents

Document	Location
TSS-WEB version 2.0 (Template)	https://tss-web.secodis.com
Information Security Policy	tbd
Password Policy	tbd

Contacts for this Document

Name	E-Mail	Section

Document History

Version	Date	Changes	Changed By
0.1	20/12/2020	Initial document based on TSS-WEB v2.0	John Doe

1 Introduction

This standard describes a baseline of security requirements for web-based applications, application components and services of Example Inc. These requirements are mandatory for newly developed applications and recommendations for existing ones. The baseline defined in this document may be increased if necessary for applications with higher protection requirements.

1.1 Scope

This standard applies for all **web development projects** that has been started after **[DD].[MM].[YYYY].** For all others of this kind, the requirements of this document are recommendations if not explicitly specified differently.

1.2 Types of Requirements

In accordance with RFC2119, two types of requirements are specified in this standard:

- **Mandatory Requirements**: Identified by terms like “MUST“, “MUST NOT”, or “HAVE TO”
- **Recommendations**: Identified by a term like “SHOULD“ or “CAN”

In case a requirement has a recommendation like nature, it does not need to be implemented if justifiable reasons exist. Recommendations that are specified with “CAN” are focused on applications of increased protection requirements or risk profile.

Exceptions to not complying with mandatory requirements must be approved by the relevant *IT security function.*

1.3 Definitions

This standard uses the following definitions:

- **3rd Party Dependency**: Here: 3rd party software artifacts, used by an *application* (e.g. libraries, Maven artifacts).
- **Application:** Here: Synonym for -> web application.
- **Change:** Every change to an application in production.
- **Confidential Data**: Data, which consists of
  (1) confidential information (e.g. trademarks, sensible business logic, passwords or personal data),
  (2) is explicitly declared as those or
  (3) is only accessible by a restricted number of people.
- **Confidential Source or Program Code:** Source or program code which may consist -> *confidential data*.
- **Criticality**: Here: Mostly synonym for business criticality.

- **Dependency Repository**: System that manages *3rd party dependencies* (e.g. libraries, Maven artifacts). A dependency repository is often part of a general software repository system such as Nexus or Artifactory.
- **External Web Application**: A web-based application that is accessible from the outside of the organization (e.g. via the Internet).
- **Internal Source or Program Code**: Source or program code which is not confidential and not public (standard).
- **Internal Web Application:** A web-based application that is only accessible from the inside of the organization (e.g. intranet application).
- **Service:** Here: Synonym for web-based service or (REST) API.
- **Source Code Repository**: System where custom code is stored (e.g. SVN, Git).
- **Web Application:** Here: A software program (UI, service or API or combination of them) that is accessible via HTTP(s) protocol and fulfills a particular business case.
- **Web-based Application:** See web application.

1.4 Roles

The following roles are referred to within this standard:

- **IT Security Function**: Organizational entity or person that is responsible for establishing and maintaining IT security requirements and checking compliance with them - e.g. by conducting security sign-offs with projects and teams. The relevant IT security function may be a security officer dedicated for a particular project or team.
- **Security Champion**: Team internal technical expert, contact and coordinator for security within a team (e.g. a development team). The responsibilities of this role include:
  (1) Security contact for a specific team.
  (2) Understands relevant security requirements and implementation/assurance of compliance to them within a team.
  (3) Identifies and manages of security risks.
  (4) Verifies correct implementation of security-relevant requirements.
  (5) Continuously verifies and improves of the effectiveness of implemented security checks and controls, it's automation and periodic assessment of security findings from tools.
  (6) Participates at internal security discussions (e.g. security Jour Fixe or communities).
- **Developer**: (Software) developers have the following responsibilities:
  (1) Has general security know-how of technologies he/she is working with and keeps it up-to-date continuously.
  (2) Capable to avoid, find and fix vulnerabilities.

- **(Development) Team**: Responsible for the security of software artifacts it develops, maintains or operates. Continuously verifies and improves the effectiveness of implemented security checks and controls, it's automation and periodic assessment of security findings from tools.

References to these roles are indicated with *italic* formatting in this standard.

1.5 Assurance Classes

Certain requirements specified in this standard depend upon two aspects of an application.

The first is the accessibility, e.g. is an application accessible from the Internet, the second is the business criticality:

		Accessibility of Application	
		INTERNAL (Not Internet-facing)	EXTERNAL (Internet-facing)
Criticality of Application	Low-Medium	**Low**	**Standard**
	High	**Standard**	**High**
	Very High	**High**	**Very High**

Table 1-1: Assurance Classes of Applications

2 Remediation of Vulnerabilities

Identified vulnerabilities in applications MUST generally be remediated quickly and causative. In case root case remediation should require a significant amount of time and the risk posed by the vulnerability is also significant, temporary measures (e.g. workarounds) SHOULD be implemented to reduce the exploitability as soon as possible until the actual root cause is fixed. Such remediation MUST always be considered as a temporary measure.

In respect of external (e.g. Internet-facing) applications, the following requirements define the point of time until a vulnerability MUST be corrected, or its exploitability prevented latest:

		Criticality of Vulnerability		
		Critical	High	Moderate
Criticality of Application	>= *High*	At the end of the next **working day**	Within **7 days**[1]	Within the next release, but after 6 months at the latest.
	<= *Moderate*	Within **7 days**	Within **21 days**	-

Table 2-1: Requirements for Vulnerability Remediation for External Applications

In respect of internal applications, the following requirements define the point of time until a vulnerability MUST be remediated if possible or at least its exploitability prevented:

		Criticality of Vulnerability		
		Critical	High	Moderate
Criticality of Application	>= *High*	Within **7 days**	Within **30 days**	Within the next release, but after 12 months at the latest.
	<= *Moderate*	Within **21 days**	Within **60 days**	-

Table 2-2: Requirements for Vulnerability Remediation for Internal Applications

[1] day = calendar day

3 Secure Operation

The following requirements apply to systems (infrastructure, platforms or other runtime environments) on which productive applications of Example Inc. are executed:

1. **Environment separation**: Systems in production MUST be strictly separated from development and test systems:
   a) Production and development MUST be separated using different environments (e.g. in cloud environments using separate accounts/subscriptions).
   b) Connections between different environments MUST not be possible.
   c) Production data SHOULD not be used on non-production systems (exceptions see chapter 6 (“Security Tests”).
   d) Users and systems MUST be authorized separately for each environment.
   e) Access servers SHOULD be separated instances for all environments but MUST be at least use separate realms.
2. **System hardening**: Systems (e.g. web servers, application servers, container platforms orcontent management systems, cloud platforms or other runtime environments) MUST be hardened according to common best practices. This includes:
   a) A hardened OS (e.g. using a hardened base image, see below),
   b) Deactivation of all services, plugins and other functionality that is not needed, especially if they are exposed (executable from remote).
   c) Hardened SSL/TLS stack in accordance to requirements of section 1 (“Service & API Security”).
   d) Activated security headers according to

e) Appendix A: Requirements for HTTP Security Header.

f) Removal of samples and other default content.

g) Execution of network services (e.g. web or application servers) with only minimal privileges and isolated from other processes if possible (e.g. as an isolated container or dedicated server instance, VM or host).

h) Network services bound to localhost if access only required from same system.

i) Network services should only be accessible from certain IPs if possible.

j) Deactivation of file handlers that are not required (e.g. ".php" for a Java application).

k) Deactivation of insecure HTTP methods (e.g. TRACE and TRACK).

l) Web and application servers must not disclose details on the server-side software stack (e.g. version numbers). Related HTTP response headers such as "Server" or "X-Powered-By" are to be deactivated or filtered.

3. **Docker security:**

   a) Docker images MUST only be build using trusted repositories.

   b) Docker images MUST only use selected base images.

   c) Docker images MUST be updating OS packages at build.

   d) Docker images MUST be scanned for insecure 3rd party components and insecure configuration.

   e) Docker images MUST not consists of a remote shell like SSH or telnet.

   f) Docker container MUST have a maximal lifetime after which they have to be rebuild with updated OS dependencies.

4. **Securing access to backend resources:**

   a) Every process MUST only have the required permissions to resources such as on the file system or database (least privilege principle). Example: "no root permission on databases".

   b) Access to backend systems MUST be authenticated and authorized in accordance to requirements of section 1 ("Service & API Security").

   c) Access to backend systems MUST use dedicated credentials for each system.

   d) Secrets MUST be securely stored and managed in accordance to requirements of section 8.11 ("Protection of Secrets").

5. **Isolation of external systems**:

   a) Applications that are directly accessible from the Internet (or other untrusted networks) MUST be deployed in an isolated network zone.

   b) All external communication MUST be encrypted using TLS/HTTPS.

   c) All external access to internal network zones MUST be approved.

   d) Outgoing communication (egress) to the Internet MUST be restricted and SHOULD be handled by proxies (e.g. HTTP proxies or SMTP proxies).

   e) Incoming communication (ingress) SHOULD be handled by reverse proxies (e.g. API gateways, Web gateways).

f) A web application firewall (WAF) MAY be used here, e.g. as an additional layer of protection, for virtual patching or as an application IDS.

6. **Administrative access** MUST be as restricted as possible:
   a) Limited to required persons only, if possible using dedicated personal accounts.
   b) Username “admin” should not be used.
   c) Limited to internal network zones or authorized IPs if possible.
   d) Using a mandatory 2nd authentication factor (such as hardware tokens, authenticator apps, X.509 client certificates) in combination with a strong user password.
   e) All administrative access should be logged in a tamperproof way.
   f) Immediately revoked after they are not required anymore (e.g. user changes organizational role).
7. **System maintenance**:
   a) Systems MUST be kept up-to-date, especially in terms of security patches.
   b) Unused applications MUST be decommissioned.
8. **Security scanning:** Productive systems MUST be periodically scanned for potential security problems. For instance:
   a) Insecure configurations
   b) Missing security patches
   c) Validity and X.509 certificates
   d) Exposed development artifacts (e.g. SVN files)
   e) Potential malware infection
9. **Security monitoring**: For applications with *assurance class* >= [HIGH] Possible security incidents MUST be continuously monitored. For instance:
   a) Potential account abuse or system compromise
   b) Failures in security controls or tests
   c) Use or assignment of critical security permissions
   d) DoS (or other) attacks
   e) Critical security findings from security scans (see above).
   f) Incident management: A consistent incident management process (including roles, responsibilities escalation procedures) MUST be implemented and followed.

4 Development Environment

1. **Securing access to the dev environment:**
   a) Access to development systems (incl. and build and deployment systems) MUST be sufficiently protected.
   b) Access to development environment MUST be restricted.
   c) Remote access to development systems MUST only be possible via a secure VPN connection and multi-factor authentication (MFA).
2. **Protection of source and program code:**
   a) Access to sensitive source and program code (code that is run in production, containing PII or other sensitive information or business logic) MUST be restricted to authorized users and revoked as soon as this user does not need it anymore (e.g. leaves the team). This includes authentication and authorization of access to cloud accounts, code repositories (e.g. SVN or Git), build systems (e.g. Jenkins or TFS), Wikis and other resources such as file systems.
   b) Sensitive source and program code MUST NOT be made available to others outside of Example Inc. (e.g. within internet forums) without explicit clearance of the relevant IT security function.
   c) The code repository SHOULD be periodically scanned for exposed secrets (e.g. X.509 private keys or API keys).
   d) Source code repositories SHOULD be regularly scanned for disclosed secrets (e.g. X.509 private keys or API keys).
3. Use of trusted repositories and 3rd party dependencies (see 5. Security within Software Development Process)

5 Security within Development Process

The following requirements relate to security activities that have to be conducted during development process for applications used in the target production environment:

1. **Roles & training:**
   a) Every developer and development team MUST comply with role description for developer and development team specified at section 1.4 (“Roles”).
   b) Every development team MUST have to appoint a security champion (see section 1.4 (“Roles”)) and a deputy. One person can fill this role for multiple teams.
   c) Every team member MUST receive general awareness and role-specific security training (e.g. secure coding training and training of secure design principles for developers).
2. **Secure by design**:
   a) Security MUST be taken into account strongly during design phase. Wherever possible, security requirements SHOULD be addressed on architecture instead of code layer.
   b) For applications with *assurance class* >= [HIGH], a security architecture MUST be documented that describes relevant security aspects, security controls a threat model of the application.
   c) High-level application security objectives MUST also be mapped to functional requirements.
   d) Decisions with severe security implications MUST be regularly questioned and discussed within the team.
3. **Security within change management & agile development**:
   a) All changes of source code MUST be committed to a source code repository (e.g. Git).
   b) For applications with *assurance class* >= [HIGH], all commits on master branches SHOULD be reviewed by a second developer (e.g. via pull or merge requests).
   c) Assessment of all functional requirements and changes (e.g. User Stories) in respect of potential security risks / impact (= “security relevance”)[2] MUST be conducted by the team before their implementation.
      i This assessment MAY be conducted informally by a team if it gained sufficient experience.
      ii Teams MAY define own criteria for security-relevance.
      iii Agile development teams SHOULD integrate corresponding criteria in their Definition of Ready (DoR) and discuss security-relevance in refinement

[2] SAFECode defines a good security indicator here, by describing security-relevant changes as “Any additions or changes in security controls and functionality”: (1) Authentication (Adding or changing an authentication method, or mechanism), (2) Authorization (Shifting the trust relationships between any components or actors in the system (change of user levels, change of data access permissions, etc or adding or changing an authorization method, or mechanism), (3) Logging, monitoring and alerting (Adding or changing application monitoring, business analytics and insight, auditing and compliance requirements or forensics) or (3) Cryptography (Adding or changing cryptographic functionality: hashing algorithms, salt, encryption/decryption algorithms, SSL/TLS configuration, key management, etc). See SAFECode paper “Tactical Threat Modeling”: https://safecode.org/wp-content/uploads/2017/05/SAFECode_TM_Whitepaper.pdf

meetings and take security efforts (e.g. for verification) into account for estimation of story.

iv For *assurance class* >= [HIGH]: If such an assessment is not conducted, a deployed application increment MUST be considered to have a potential high-security risk.

v Threat models and *assurance class* MUST be reviewed and updated if affected (e.g. in case of changes of security controls or architectural change in general).

vi A suitable acceptance criteria (e.g. review by security champion, update of security documentation) MUST be defined for all security-relevant requirements and changes. Agile development teams SHOULD integrate corresponding criteria in their Definition of Done (DoD).

4. **Secure build & deployment:**
   a) A formal definition of the build & deployment process MUST be created so that it becomes consistent, repeatable and automated.
   b) Access to build and deployment systems MUST be secured according to requirements in chapter 4 (“Development Environment”).
   c) Automated security checks MUST be integrated into build & deployment processes in accordance to requirements in chapter 6 (“Security Tests”).
   d) Secrets SHOULD be injected during deployment process in accordance to requirements in chapter 8.11 (“Protection of Secrets”).
   e) Deployment pipelines SHOULD implement a pull-based model[3].
   f) For *assurance class* >= [HIGH], integrity of deployed artifacts MUST be automatically verified (e.g. using digital signatures).
   g) For *assurance class* >= [HIGH], executed scripts in build systems SHOULD be authorized and whitelisted.
5. **Security of 3rd party dependencies** in target production environment:
   a) 3rd party dependencies SHOULD only be obtained via internal and approved repositories.
   b) Before a new 3rd party dependency is allowed to be used in productive applications (or within the release build environment), it MUST be approved by the architecture board. This does not affect new releases of a dependency that has already been approved.
   c) 3rd party dependencies SHOULD be updated regularly.
   d) 3rd party dependencies MUST be updated in case of relevant critical security vulnerabilities or end of life.
   e) A Software Bill of Materials (SBOM) that keeps a record of all 3rd party dependencies used in production SHOULD be maintained.
   f) Testing requirements for custom and 3rd party code are defined at 6. Security Tests.
6. **Security approvals (security gates):**

[3] See https://www.weave.works/blog/why-is-a-pull-vs-a-push-pipeline-important

a) Initial Project approval (mandatory): All new projects that are either implementing new applications or that plan to change existing ones MUST be approved by the relevant IT security function before they are allowed to be started. As part of this approval, the relevant IT security function will specify the assurance class with the project and may define security controls that have to be implemented or security activities that have to be conducted by the project.

b) Architecture approval (conditional): For all new applications with *assurance class* >= [HIGH], or if explicitly requested by the IT security function during the project approval, the solution architecture (including security architecture that describes security controls & aspects and a threat model describing relevant threats and mitigations for them) MUST be approved by the relevant IT security function before initial implementation is allowed to begin. The IT security function MAY request this approval to be renewed for architectural changes when certain criteria are met.

c) Go-Live approval (conditional): Initial application releases for applications with *assurance class* >= [HIGH] MUST pass a security sign-off by the relevant IT security function before they are allowed to be used in the target production environment. The relevant IT security function MAY decide within the project approval as well that this approval is required for subsequent releases (e.g. based on certain criteria) or for projects with a lower assurance class.

d) All security approvals and risks management decisions must be documented.

7. **Remediation of security findings** with a criticality >= [HIGH] (or CVSSv3[4] Score >= 7.0) MUST be sufficiently mitigates before a new application release is allowed to go-live:

   a) In case this is not possible, the relevant risk MUST be accepted by the respective management function (e.g. project lead). For applications with *assurance class* >= [HIGH], this risk acceptance MUST be formally documented (e.g. as Jira ticket).

   b) In addition, security findings with criticality >= [MEDIUM] (or CVSS v3 Score >= 6.0) SHOULD NOT go-live without proper verification.

   c) Teams MAY reassess tool findings. For instance, they may refine a CVSS Base Score by evaluating its CVSS Environmental Score and thereby taken aspects like its classification or accessibility into account. When a rating/score is refined the respected reason (e.g. CVSS vector) MUST be documented.

   d) Identified vulnerabilities MUST be retested after remediation to verify that countermeasures has been implemented correctly.

   e) For *assurance class* >= [HIGH]: Exceptions (such as temporary workarounds) MUST be approved by the IT security function.

8. **Security documentation:**

   a) A comprehensive security documentation MUST exist and formally be approved by the relevant IT security function for every application with *assurance class* >= [HIGH] before its implementation starts (relevant aspects) and it's allowed to initially go-live (complete documentation).

[4] CVSS = Common Vulnerability Scoring System (CVSS) v3, https://www.first.org/cvss

b) In case the relevant IT security function has not requested it differently, this document SHOULD cover the following aspects:
    i Data and application classification (assurance class),
    ii application security architecture (relevant application components, interfaces and relevant data flows and architectural security controls in the form of a diagram),
    iii network/cloud security architecture,
    iv list of relevant security requirements (e.g. security standards or business security requirements),
    v known threats and respective countermeasures to mitigate them,
    vi role and authorization concept (required for go-live),
    vii data handling that describes what data handles where and how (required for go-live),
    viii operational security controls (required for go-live).

6 Security Tests

The following requirements relate to security testing activities that have to be carried out within the development process of productive applications developed within Example Inc.:

1. **General requirements:**
   a) Correct and complete implementation of security & security-relevant requirements and changes MUST be verified with suitable security tests.
   b) Where possible, security tests SHOULD be executed automatically and continuously (e.g. within CI and CD pipelines) and as early as possible within the development process (fail fast principle).
   c) Test data MUST not contain confidential or personal (PII data) references.
   d) The execution of security tests MUST NOT be affected by perimetric security systems (e.g. a web application firewall).
2. **Defect tracking:**
   a) Security defects MUST be tracked in defect tracking system and provide relevant meta information (e.g. criticality rating, CVSS score etc.).
   b) Open security defects SHOULD be regularly checked for relevance and possible quick wins.
   c) For applications with *assurance class* >= [HIGH], security risks MUST also tracked in defect tracking system.
3. **Utilize automated security testing tools:**
   a) Applications MUST be automatically analyzed with security code scanning tools (SAST or IAST) in order to identify implementation vulnerabilities in source and program code as early as possible.
   b) Applications MUST be automatically analyzed with SCA (Software Composition Analysis) tools for any known vulnerabilities in 3rd party dependencies.
   c) Docker images used in the target production environment MUST be automatically scanned for security issues.
   d) Security-relevant configuration MUST be automatically scanned for security issues.
   e) For applications with *assurance class* >= [HIGH] Security testing tools MUST automatically enforce security requirements and security testing policies within the build process as specified at chapter 5 (“Security within Development Process”).
   f) Applications with assurance class [VERY HIGH] SHOULD be periodically tested for denial-of-service attackability.
4. **Custom security tests & developer tests:**
   a) Security tests of implemented security requirements and security controls (e.g. authentication or access controls) SHOULD be implemented, executed, preferably in an automatically and continuously way.

b) Developer and system acceptance testing SHOULD include testing for functional (security controls) and non-functional[5] security requirements.

c) Abuse cases SHOULD be created & tested for critical business security aspects in applications with *assurance class* >= [HIGH].

5. **Pentests:**

a) Applications MUST be verified by penetration tests according to the pentesting policy bellow. Penetration tests SHOULD be carried out within environments that are close to production environments (e.g. on integration systems).

b) After an severe vulnerability has been fixed, a retest should be executed, ideally by the same tester, in order to verify the correct implementation of the fix.

		Accessibility of Application	
		External Application (Internet-Facing)	*Internal Application* (Non-Internet-Facing)
Business Criticality of Application	>= *High*	Before initial go live but at least **annually**[6]	ASAP after initial go-live but at least every **third year**
	<= *Medium*	Before initial go live but at least every **second year**	-

Table 6-1: Pentesting Policy

[5] E.g. using predefined tests for common injection vulnerabilities (XSS, SQLi), access controls (trying to access sensitive objects with no/insufficient privileges) and fuzz testing.

[6] In the case where it is ensured that no changes are made to an application, this interval MAY be extended by one additional year.

7 Oitsourced Development

The following requirements relate to contractors that implement applications on behalf of Example Inc. and in addition all other requirements of this document if applicable (e.g. protection of source code). Every supplier MUST comply with the following requirements:

1. Due diligence: Implementation of all measures and common best practice within the development, operation and quality assurance required to prevent the occurrence of new security defects.
2. For applications with *assurance class* >= [HIGH]:
3. Evidence[7]that security has been taken into account throughout the development process.
4. A security concept that complies to the security documentation requirements specified in chapter 5 (“Security within Development Process”).
5. Implementation of all necessary and requested security measures in order to reach a suitable level of protection, including those listed in chapter **Fehler! Verweisquelle konnte nicht gefunden werden.** (“**Fehler! Verweisquelle konnte nicht gefunden werden.**”) and 8 (“Implementation Requirements”).
6. Appointment of an internal contact person for security-relevant questions.
7. Only persons have access to the source code created on behalf of Example Inc. that are authorized and required (need to know principle).
8. Right to audit: On its own discretion, Example Inc. is allowed to conduct security assessments of source code and applications that have been created on its behalf. The supplier will provide the required support if needed.
9. Security vulnerabilities are to be remediated as soon as possible without extra costs when requested by Example Inc. (see relevant requirements in chapter 2 ("Remediation of Vulnerabilities”)).
10. Security SHOULD be built into supplier agreements in order to ensure compliance with organizational requirements.

[7] e.g. via ISO 27001 certfication and/or BSIMM vor Vendors (https://www.bsimm.com/about/bsimm-for-vendors)

8 Implementation Requirements

The following requirements the implementation of web applications of Example Inc.:

8.1 General Principles

1. *Minimize Attack Surface*: Interfaces, functionality, parameters, services, and protocols that are not required MUST be disabled on external and SHOULD be disabled on internal systems.
2. *Don't Trust User Input*: Data that has been received from an untrusted source (e.g. a web browser) MUST generally be mistrusted and therefore strongly validated.
3. *Defense*-in-Depth: Security SHOULD be implemented based on multiple layers to compensate for ineffective security controls.
4. *Keep Security Settings Adaptable*: Security parameterization SHOULD be declared where possible (= with configuration statements or annotations) instead of programmable (= with program code) where possible.
5. *Externalize Security Functions:* Security functions SHOULD be externalized (e.g. using an external authentication system).
6. *Keep Security Consistent:* Identical security controls (e.g. one within the web frontend and another within an AJAX interface) SHOULD be implemented with the same security function (= program code) and policy.
7. *Use Mature Security Controls*: Security relevant program code SHOULD only be implemented with mature and tested technologies, algorithms and implementations (APIs, frameworks etc.)
8. *Keep Security Testable*: Before a new technology (protocol, framework, API, etc.) is being used in production, it SHOULD be ensured that proper testing procedures and tools exist for performing security tests on them.
9. Use Secure Defaults: Use secure / safe defaults (e.g. in frameworks) to prevent unintentional security problems.

8.2 Input Validation

1. All untrusted input parameters (e.g. of external interfaces) MUST be validated restrictively.
2. Security-relevant input validation MUST be performed on the server-side. Client-side validation MAY be implemented but only for usability reasons in addition to server-side validation or to prevent client-side attacks.
3. Input validation SHOULD be performed as strictly as possible in respect of allowed data type, length, and range. Examples:
    a) Numeric instead of a String datatype,
    b) limitations for numeric data types or
    c) restricted allowed characters for a string (e.g. only "a-z" and "A-Z").
4. Input validation MUST be applied to all types of untrusted input parameters (including hidden form fields and HTTP header such as cookies).

5. In order to remove path truncations like "../../", directory paths MUST be normalized / before input validation is applied to it[8]. This MAY be done implicitly by using a secure API that removes such truncations automatically.
6. Input validation SHOULD be using a positive validation model (whitelisting).
7. Validation of user input SHOULD be performed implicitly with data binding (typecasting) where possible.
8. Validation of non-editable application parameters (no user input) SHOULD be performed implicitly via integrity checks or indirection mappings where possible.
9. HTML input MUST be validated restrictively with a mature HTML sanitizer API.
10. JSON or XML data from untrusted sources (e.g. received by a service) MUST be validated indirectly (e.g. via bean validation) or directly via an XML Schema. This MUST be done restrictively as described above.
11. An XML parser that process XML content from untrusted sources (e.g. from an external entity) MUST be hardened to prevent common XML-based attacks:
    a) Set restrictive limits (e.g. in respect of maximal nesting depth or document size),
    b) deactivate processing of external XML entities.
12. In order to prevent insecure object deserialization, it MUST be ensured that objects received and bound from untrusted sources are not hostile or tempered (e.g. by only binding non-sensitive attributes, perform whitelisting or integrity checks.

8.3 File Uploads and Downloads

1. **Authentication**: All file uploads SHOULD be authenticated by the user on the server-side (= function only available within the authenticated user area) and MUST implement suitable controls to preventing denial of service attacks if available to not authenticated users.
2. **Storage**:
    a. Uploaded files SHOULD be stored to a database.
    b. In case uploaded files are required to be stored on a file system, the implementation MUST be according to the following requirements:
        i Uploaded files MUST be stored in an access protected directory that is not accessible from external (e.g. stored outside of the web & document root). For applications with *assurance class* >= [HIGH], uploaded files SHOULD be stores in a user-specific directory.
        ii File uploads MUST be saved with restrictive permissions (e.g. in case of Unix-based systems „`chmod 0600`").
        iii Uploaded files MUST NOT be executable.
        iv Uploaded files MUST be stored as a new file with unique filename that is not user-specified.
3. **Limitation**:
    a) The size of uploaded files MUST be restricted to a reasonable maximum (e.g. 5 MB).
    b) The number of files that are allowed to be uploaded SHOULD be restricted to a reasonable value (e.g. 8 files from one user per hour and user or IP address).
4. **Validation**:

[8] Often APIs like `getCanonicalPath()` exists for this matter.

a) File types that may consist of executable code (e.g. „.html“, “.js“ or „.exe“) MUST be prevented from being uploaded.
b) File type validation MUST be executed based on a whitelisting approach where only safe file types are allowed.
c) File type validation MUST be executed on file extension and MIME type.
d) For applications with *assurance class* >= [HIGH], the file type SHOULD be verified with suitable tools or APIs based on its actual content (e.g. you could perform image operations like `getImageSize()` on an expected image file).

5. **Sanitization**: Uploaded files from untrusted sources (e.g. over the Internet) with potentially executable content SHOULD be
   a) sanitized of executable code (e.g. macros in MS Word files) and
   b) analyzed with an antivirus (AV) solution for potential malware infections. Files MUST be rejected or sent to quarantine in cases of a positive finding.[9]
6. **Download**:
   a) File downloads SHOULD be carried out from a separate origin (e.g. „files.example.com“), to prevent the impact of executable script code that they may contain.
   b) File downloads SHOULD always be protected with relevant HTTP security headers, e.g. to prevent MIME Type sniffing in browsers (see Appendix A: Requirements for HTTP Security Header).
   c) Filenames MUST be properly encoded before written into response header when user-controlled in order to preventing Reflected File Downloads (RFD).

8.4 Output Validation (Encoding & Escaping)

2. Every access to backend systems such as databases MUST be parameterized[10], e.g. via prepared statements, OR mappers (e.g. Hibernate) when an API for this matter does exist.
3. All parameters, either internally or user-controlled, MUST be declared as parameters within a prepared statement (or similar API) call.
4. In case an API does not provide any methods for parameterized statements, parameters MUST be encoded with suitable APIs (e.g. SQL encoding) to prevent interpreter injection.
5. User-controlled parameters MUST be encoded with a suitable API and method related to its output context if written into a webpage:
   a) HTML context: HTML entity encoding
   b) JavaScript context: JavaScript escaping
   c) CSS context: CSS escaping
6. Output encoding SHOULD only be implemented with mature APIs and frameworks.
7. Implicit validation (e.g. via ORM frameworks or template technologies) SHOULD be preferred to explicit validation (API calls).

[9] This function can be tested with the EICAR test file (www.eicar.org)
[10] Each parameter needs to be defined explicitly as a parameter (e.g. with a `setParam()` statement) or indirectly by a framework.

8.5 User Authentication and Registration

1. User registration MUST be implemented with an identification method that is suitable for the *assurance class* of the application it is used for:
    a) <= [STANDARD]: E-mail addresses MAY be used.
    b) [HIGH]: An additional factor (e.g. mobile phone) SHOULD be used.
    c) [VERY HIGH]: Personal identification is required.
2. Users MUST NOT be allowed to log into the application before their identification process has been completed.
3. Registration and authentication of external users MUST be implemented on external applications in a way that they prevent automated attacks (e.g. brute forcing). Examples are delays or CAPTCHAs.
4. User authentication MUST be implemented with suitable mechanisms in respect of its *assurance class*:
    a) <= [STANDARD]: Password-based authentication with secure methods and protocols. Password must be compliant to the password policy (NIST Authenticator Assurance Level 1[11]).
    b) [HIGH]: Same requirements as for standard but with an additional authentication factor. This factor must be exchanged using a separate and secure channel or layer of communication. It may be stored on the same system where the password is entered through (soft crypto token). Examples are X.509 certificates, one-time tokens that are sent via SMS or e-mail, time-based tokens (e.g. via Google Authenticator App), or IP addresses. (NIST Authenticator Assurance Level 2).
    c) [VERY HIGH]: Same as High but with the authentication factor required to be generated on a separate system (hard crypto token) that has been approved for this purpose. Examples are RSA SecureID tokens (NIST Authenticator Assurance Level 3).
5. HTTP Basic authentication SHOULD only be used as additional access protection and MUST always be used with HTTPS.
6. Usernames SHOULD be personalized.
7. In case a user login fails, the application MUST NOT disclose information on the cause of the error (e.g. “password is wrong”). Positive example: “Username or password was incorrect.”.
8. Autocomplete for all fields on login dialogs SHOULD be deactivated by using the following statement `autocomplete="off"`.

8.6 User Passwords I: Strength and Usage

1. User password MUST be compliant to the password policy at both registration as well as password change and password reset functions.
2. As long as not specified differently by the password policy, the following requirements for user passwords MUST apply:

[11] See NIST SP 800-63b, NIST Special Publication 800-63B Digital Identity Guideline, Authenticator Assurance Levels: (https://pages.nist.gov/800-63-3/sp800-63b.html#sec4)

a) Length >= 8 characters,
b) consists of characters, digits and special characters,
c) not be identical to the username,
d) be masked on all HTML password fields,
e) not be logged or cached,
f) encrypted when transferred over insecure channels,
g) not transmitted in URLs and
h) stored as a salted secure hash, ideally with key stretching. This SHOULD be implemented with bcrypt, scrypt, PBKDF2 or Argon2 algorithm.

3. Initial user passwords MUST be changed by the user at first login.
4. Standard password (= set by the vendor) MUST NOT be used and replaced by strong individual passwords.
5. **Password change functions**:
   b) Users MUST be able to change their passwords.
   c) Users SHOULD be indicated the strength of the current password choice when changing their passwords (using a password strength functions).
   d) Users MUST confirm a new password with their current ones.
   e) Users SHOULD be informed when their password has changed (e.g. via e-mail notification).
6. **Password forgot functions**:
   b) MUST implement the same level of security protections as the user authentication function (e.g. anti-automation).
   c) MUST be authorized by the user with the same method that is used as second factor or (in case no second factor is used) for user identification (e.g. e-mail address). Ideally, by using a One Time Token (OTT) with limited validity sent as a second factor (e.g. to the registered user e-mail address or mobile phone).
   d) MUST not affect the state of the user profile before password reset is completed.

8.7 Hardening of Session Management

Is the session management implemented with certain standard components or services? Then reference them here.

1. The session management MUST be based on the standard implementation of the application server or web container.
2. Session IDs MUST be
   a) at least 120 bit strong,
   b) generated with a cryptographically secure pseudorandom number generator (CSPRNG) and be completely random,
   c) transferred encrypted (via TLS/HTTPS) on insecure networks
   d) renewed after every successful user authentication,
   e) NOT be transmitted in URLs.

3. Session cookies MUST be restricted in respect of their validity:
   a) Set both security attributes „`httpOnly`", „`secure`" and „`SameSite`"
   b) Avoid persistent cookies (don't set expire attribute)
   c) Set a path attribute to the base URL in case multiple applications are operated on the same system.
4. Authenticated server-side sessions:
   a) MUST be invalidated after a user has been logged-in successfully,
   b) MUST be invalidated after an authenticated user has been idle for more than 30 minutes (idle or soft logout),
   c) SHOULD be invalidated after a user session has been active more than 24 hours,
   d) SHOULD only exist once per user. When a user logs on, all existing session object of this user SHOULD be invalidated.
5. All state-changing operations (create, update, delete) on an authenticated user session MUST be protected against session replay and Cross-Site Request Forgery (CSRF).
   a) State-changing operations MUST be protected with a cryptographic random replay token (e.g. as an additional Hidden Fields or as X-header) that is unique for the user session or a specific request and for instance.
   b) State-changing operations MUST be denied if a CSRF token is invalid or missing.
   c) If a web framework provides an own CSRF protection mechanism, then this SHOULD be used.
   d) State-changing operations MUST NOT be possible via HTTP GET.

8.8 Access Controls

Are standard components used for authenticating users (policy decision point) or retrieving user roles and permissions (policy information point)? Then they should be described here.

1. Every sensitive object access (e.g. access to a sensitive object within the database) MUST be authorized on the server-side (complete mediation).
2. Access controls SHOULD be applied on different layers if possible (e.g. URL, file, method and object layer) or via an indirection layer to reduce the risk of insecure object references.
3. Access controls MUST not only verify if the requesting entity has all required roles for specific access but also if this particular entity has the required permission to access a specific data object.
4. Every process and role SHOULD be implemented as restrictive as possible according to its particular business requirement.
5. For technical services/APIs access in accordance to requirement from 8.13 ("Service & API Security").

8.9 Error Handling & Logging

1. It MUST be ensured that in the event of any error (expected or unexpected) the application stays in a secure state in which that no internal information (such as stack traces) is disclosed to users.
2. Security exceptions SHOULD be thrown in case of security failures.
3. Security-relevant events (e.g. user log-ins, unauthorized access to sensitive data, sensitive actions on a user profile, potential misuse or attacks) MUST be logged.
4. The following information SHOULD be logged as part of a security event:
    a) Security tag (“SEC”)
    b) Timestamp
    c) Subject (e.g. user ID, source IP)
    d) Event description (e.g. sensitive access / change to object)
    e) Relevant component
    f) Result
5. Technical logs MUST not contain Personal Identifiable Information (PII).

8.10 Data Security & Cryptography

In the case of cryptographic requirements exist, they should be referenced here.

1. Only standard and mature cryptographic algorithms, operation modes, key lengths ciphers, and implementations MUST be used.
2. **General Transmission**
    a) Transmission of sensitive data SHOULD generally only be possible via TLS/HTTPS.
    b) In cases where access requires HTTPS, requests via HTTP MUST be redirected to HTTPS. This SHOULD be implemented with a permanent redirection (HTTP 301).
    c) HTTPS servers MUST only support current secure ciphers and protocols. Insecure ones (e.g. SSLv2 and RC4 cipher) MUST be deactivated.
    d) Confidential data MUST only be sent within the HTTP Request Body (e.g. via HTTP POST) but not within URLs (exception: object IDs).
3. **Transmission on untrusted channels** (e.g. the Internet) MUST
    a) only be possible with HTTPS using valid certificates.
    b) using HTTP Strict Transport Security (HSTS) headers 8.12 Client-Side Security and
    c) only be sent with anti-caching response headers (see Appendix A: Requirements for HTTP Security Header).
4. **Encryption at Rest**
    a) Confidential data MUST be encrypted before stored on the client-side or on external cloud environments.
    b) User passwords MUST be persisted with suitable methods (see section 8.6 User Passwords).
5. **X.509 Certificates**
    a) External HTTPS connections MUST use valid X.509 certificates issued by a trusted authority (CA).

b) X.509 certificates MUST use RSA with >= 2048 bit or ECC with >= 256 bit.

c) External customer applications (UIs) SHOULD use EV certificates.

8.11 Protection of Secrets

The following requirements are relevant for secrets (e.g. passwords of technical users, API keys, credentials or private keys) used either in production or that are generally used to protect access to sensitive data:

1. Secrets MUST be encrypted when stored with the source code (otherwise they need to be separated from it).
2. Access to secrets MUST be restricted.
3. For *assurance class* >= [HIGH]
   a) Secrets MUST be stored in a secure secret store (vault) or keystore)
   b) Secrets MUST be stored encrypted
   c) Secrets SHOULD be rotated at least one per year.

8.12 Client-Side Security

1. Use of secure JavaScript APIs:
   a) JSON code MUST only be parsed with a secure JavaScript API such as `JSON.parse()` (not `eval()`).
   b) Instead of unsafe JavaScript APIs that do allow to write HTML code directly (e.g. „`.innerHTML`“), safe APIs SHOULD be used that only write text output (e.g. „`.innerText`“ or „`.textContent`“). The same requirement applies to web frameworks that provide such functionality.
2. HTTP header of Web UIs MUST be implemented according to Appendix A: Requirements for HTTP Security Header.
3. Only confidential user data MAY be stored on the client-side but this SHOULD only be be in an encrypted way.
4. User states or other meta-information MUST only be stored with sufficient integrity protection (e.g. as a signed JWT token).

8.13 Service & API Security

Are certain kinds of security functions implemented with existing security services such as XML firewalls or access gateways within the organization? Then these components should be specified here.

1. External services SHOULD only be made available externally using a hardened Service/API Gateway.
2. **Authentication:**
   a) The strength of the authentication mechanism used MUST be adequate to the protection needs.
   b) Shared secrets used for services authentication (e.g. API keys or OAuth 2.0 Client Secrets) MUST have the following characteristics:

i. Length min 32 characters (= 256 bit)
ii. Cryptographically random (consisting of alpha-numeric and special characters)
iii. Stored securely according to the requirements specified in
iv. Transmitted outside of URLs (e.g. via HTTP POST or HTTP header).

c) For *assurance class* >= HIGH, external service-to-service communication SHOULD be authenticated using asymmetric cryptography (e.g. X.509 certificates or signed JWT access tokens).

d) Authentication credentials MUST be unique for different systems and environments (e.g. test, production).

3. **Access Tokens:**

a) Services with assurance class >= HIGH SHOULD be protected using Access Tokens (OAuth 2.0 or SAML)

b) Following requirements do apply to access tokens:

i. Short-lived
ii. Restrictive scope
iii. Transmitted only via HTTPS
iv. Issued by a trusted and hardened server (e.g. OIDC Identity Server or OAuth 2.0 Authorization Server)
v. created and validated using mature APIs

4. **OAuth 2.0/OICD** requirements:

a) Only 3-Legged: Authorization Code Grant with PKCE (or alternative authorization code binding technique, e.g. “state” binding or OICD Nounces) for both public clients (e.g. single-page applications (SPAs)[12] SHOULD be used for both public and confidential clients (e.g. server-side web apps).[13]

b) Only 3-Legged: For CSRF protection, Authorization Code Grant Flows MUST either use the “state” parameter, PKCE, or “OICD Nounces

c) Only 3-Legged: Clients MUST register (and be validated using) full redirect URI as described in RFC 6819 Section 5.2.3.5 (no pattern matching).

d) Service-to-service access SHOULD be protected using Client Credential Grant.

e) HTTPS MUST be used for all communication.

f) See requirements for Access Tokens above.

5. **Web UI APIs:**

a) MUST implement the same security requirements for access to external resources as the corresponding web UI (authentication, validation, etc.).

b) MUST not contain script code (e.g. JavaScript) that is directly executable.

c) MUST contain CSRF protection if working in user context (see section 8.7 Hardening of Session Management).

[12] See https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps-00
[13] See https://tools.ietf.org/html/draft-ietf-oauth-security-topics-14

6. **Cross-Domain Access:**
    a) Cross-domain requests MUST be performed using secure mechanisms such as Cross-Origin Resource Sharing (CORS) and a restrictive policy (e.g. restricted so certain hosts or domains or with credential submission deactivated).
    b) The origin header of cross-domain requests MUST be authorized on the server-side.
7. Services SHOULD implement restrictive limits (max. requests per client or time interval).
8. WebSockets MUST transfer all confidential data with wss:// schema and an additional server-side authorization of the source origin header.

Appendix A: Requirements for HTTP Security Header

Modern browsers support several protection mechanisms that can be activated and configured by a web application via HTTP response headers. The table below describes related requirements and recommendations for external web applications in production:

Headers that are centrally included:

Response Header	General Requirement	For Web UIs?	For APIs?
Content-Type	`...; charset=utf-8`	Yes	Yes
Strict-Transport-Security[14]	`max-age=10886400; includeSubDomains; preload`	Yes (if HTTPS)	Yes
X-XSS-Protection[15]	`1; mode=block`	Yes	No (but does no harm if yes)
X-Frame-Options	`SAMEORIGIN`	Yes	No
Referrer Policy	`same-origin`	Yes	No (but does no harm if yes)
X-Content-Type-Options[16]	`nosniff`	Yes	No (but does no harm if yes)

Headers that must be set within the Web application:

Response Header	General Requirement	For Web UIs?	For APIs?
Set-Cookie	`… ;httpOnly; secure; SameSite`	Yes, when they transfer of confidential data in cookies	Yes, when they transfer of confidential data in cookies
Cache-Control	`no-cache, no-store`	Whenever confidential data is transmitted.	Whenever confidential data is transmitted.
Pragma	`no-cache`		
Expires	`-1`		
Content-Security-Policy[17]	`object-src 'none'; script-src 'self' [URL1] [URL2]; style-src 'self' unsafe-`	General recommendation for new for all Web UIs.	No

[14] HTTP Strict Transport Security (HSTS) forces users of a web site to exclusively access it via HTTPS for a defined amount of time. Before using this header, you should ensure that all requests to this host (and to all subdomains when using „includeSubDomains" attribute) can be executed using HTTPS. This header prevents certain man-in-the-middle attacks.

[15] Additional Cross-site Scripting prevention (IE only).

[16] Deactivation of MIME sniffing in browsers that are used to identify MIME types but also to execute certain Cross-site Scripting attacks.

[17] A Content Security Policy (CSP) provides an additional but very effective client-side protection against many common client-dite attacks such as Cross-site Scripting (XSS). Applications must explicitly support it which often requires a CSP compliant MVC framework.

	`inline; object-src 'self';base-uri 'none';`	Not required for APIs.	
	`object-src 'none'; script-src 'nonce-{random}' 'unsafe-inline' 'unsafe-eval' 'strict-dynamic' https: http:; base-uri 'none'; report-uri https://your-report-collector.example.com/`	Recommendation for new Web UIs that have to use inline script blocks (e.g. if integrated by a JS framework). Do not use this setting if you do not have to since it disables CSP protection for older browsers!	No (but does no harm if yes)
Content-Disposition	`attachment; filename=<filename>`	Web UIs at which users can download files that are potentially untrusted.	No (but does no harm if yes)
X-Download-Options	`noopen`		No (but does no harm if yes)

<u>Caution</u>: Settings these headers may have implications on the proper functionality of a web application. Therefore, activating a new header SHOULD always be combined with comprehensive functional tests.