NIST IR 8477-Based Set Theory Relationship Mapping (STRM)
Reference document: Secure Controls Framework (SCF) version 2026.1
STRM Guidance: https://securecontrolsframework.com/start-here/set-theory-relationship-mapping-strm/

Focal Document: OWASP Top 10 - 2025
Focal Document URL: https://owasp.org/Top10/2025/
Published STRM URL: https://content.securecontrolsframework.com/strm/scf-strm-general-owasp-top-10-2025.pdf

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Continuous Monitoring	MON-01	Mechanisms exist to facilitate the implementation of enterprise-wide monitoring controls.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Intrusion Detection & Prevention Systems (IDS & IPS)	MON-01.1	Mechanisms exist to implement Intrusion Detection / Prevention Systems (IDS / IPS) technologies on critical systems, key network segments and network choke points.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Automated Tools for Real-Time Analysis	MON-01.2	Mechanisms exist to utilize a Security Incident Event Manager (SIEM), or similar automated tool, to support near real-time analysis and incident escalation.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Inbound & Outbound Communications Traffic	MON-01.3	Mechanisms exist to continuously monitor inbound and outbound communications traffic for unusual or unauthorized activities or conditions.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	System Generated Alerts	MON-01.4	Mechanisms exist to generate, monitor, correlate and respond to alerts from physical, cybersecurity, data protection and supply chain activities to achieve integrated situational awareness.	3	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	File Integrity Monitoring (FIM)	MON-01.7	Mechanisms exist to utilize a File Integrity Monitor (FIM), or similar change-detection technology, on critical Technology Assets, Applications and/or Services (TAAS) to generate alerts for unauthorized modifications.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Security Event Monitoring	MON-01.8	Mechanisms exist to review event logs on an ongoing basis and escalate incidents in accordance with established timelines and procedures.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Analyze and Prioritize Monitoring Requirements	MON-01.16	Mechanisms exist to assess the organization's needs for monitoring and prioritize the monitoring of Technology Assets, Applications and/or Services (TAAS), based on TAAS criticality and the sensitivity of the data it stores, transmits and processes.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Protection of Event Logs	MON-08	Mechanisms exist to protect event logs and audit tools from unauthorized access, modification and deletion.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Embedded Technology Security Program	EMB-01	Mechanisms exist to facilitate the implementation of embedded technology controls.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Restrict Access To Security Functions	END-16	Mechanisms exist to ensure security functions are restricted to authorized individuals and enforce least privilege control requirements for necessary job functions.	3	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Identity & Access Management (IAM)	IAC-01	Mechanisms exist to facilitate the implementation of identification and access management controls.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	User Provisioning & De-Provisioning	IAC-07	Mechanisms exist to utilize a formal user registration and de-registration process that governs the assignment of access rights.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Change of Roles & Duties	IAC-07.1	Mechanisms exist to revoke user access rights following changes in personnel roles and duties, if no longer necessary or permitted.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Termination of Employment	IAC-07.2	Mechanisms exist to revoke user access rights in a timely manner, upon termination of employment or contract.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Role-Based Access Control (RBAC)	IAC-08	Mechanisms exist to enforce Role-Based Access Control (RBAC) for Technology Assets, Applications, Services and/or Data (TAASD) to restrict access to individuals assigned specific roles with legitimate business needs.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Identifier Management (User Names)	IAC-09	Mechanisms exist to govern naming standards for usernames and Technology Assets, Applications and/or Services (TAAS).	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	User Identity (ID) Management	IAC-09.1	Mechanisms exist to ensure proper user identification management for non-consumer users and administrators.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Identity User Status	IAC-09.2	Mechanisms exist to identify contractors and other third-party users through unique username characteristics.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Dynamic Management	IAC-09.3	Mechanisms exist to dynamically manage usernames and system identifiers.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Privileged Account Identifiers	IAC-09.5	Mechanisms exist to uniquely manage privileged accounts to identify the account as a privileged user or service.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Pairwise Pseudonymous Identifiers (PPID)	IAC-09.6	Mechanisms exist to generate pairwise pseudonymous identifiers with no identifying information about a data subject to discourage activity tracking and profiling of the data subject.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Account Disabling for High Risk Individuals	IAC-15.6	Mechanisms exist to disable accounts immediately upon notification for users posing a significant risk to the organization.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Privileged Account Inventories	IAC-16.1	Mechanisms exist to inventory all privileged accounts and validate that each person with elevated privileges is authorized by the appropriate level of organizational management.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Periodic Review of Account Privileges	IAC-17	Mechanisms exist to periodically-review the privileges assigned to individuals and service accounts to validate the need for such privileges and reassign or remove unnecessary privileges, as necessary.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Access Enforcement	IAC-20	Mechanisms exist to enforce Logical Access Control (LAC) permissions that conform to the principle of "least privilege."	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Access To Sensitive / Regulated Data	IAC-20.1	Mechanisms exist to limit access to sensitive/regulated data to only those individuals whose job requires such access.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Database Access	IAC-20.2	Mechanisms exist to restrict access to databases containing sensitive/regulated data to only necessary Technology Assets, Applications and/or Services (TAAS) or those individuals whose job requires such access.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Use of Privileged Utility Programs	IAC-20.3	Mechanisms exist to restrict and tightly control utility programs that are capable of overriding system and application controls.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Least Privilege	IAC-21	Mechanisms exist to utilize the concept of least privilege, allowing only authorized access to processes necessary to accomplish assigned tasks in accordance with organizational business functions.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Authorize Access to Security Functions	IAC-21.1	Mechanisms exist to limit access to security functions to explicitly-authorized privileged users.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Non-Privileged Access for Non-Security Functions	IAC-21.2	Mechanisms exist to prohibit privileged users from using privileged accounts, while performing non-security functions.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Management Approval For Privileged Accounts	IAC-21.3	Mechanisms exist to restrict the assignment of privileged accounts to management-approved personnel and/or roles.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Auditing Use of Privileged Functions	IAC-21.4	Mechanisms exist to audit the execution of privileged functions.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Prohibit Non-Privileged Users from Executing Privileged Functions	IAC-21.5	Mechanisms exist to prevent non-privileged users from executing privileged functions to include disabling, circumventing or altering implemented security safeguards / countermeasures.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Session Lock	IAC-24	Mechanisms exist to initiate a session lock after an organization-defined time period of inactivity, or upon receiving a request from a user and retain the session lock until the user reestablishes access using established identification and authentication methods.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Pattern-Hiding Displays	IAC-24.1	Mechanisms exist to implement pattern-hiding displays to conceal information previously visible on the display during the session lock.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Session Termination	IAC-25	Automated mechanisms exist to log out users, both locally on the network and for remote sessions, at the end of the session or after an organization-defined period of inactivity.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Permitted Actions Without Identification or Authorization	IAC-26	Mechanisms exist to identify and document the supporting rationale for specific user actions that can be performed on a system without identification or authentication.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Data Privacy Requirements for Contractors & Service Providers	PRI-07.1	Mechanisms exist to include data privacy requirements in contracts and other acquisition-related documents that establish data privacy roles and responsibilities for contractors and service providers.	5	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Secure Engineering Principles	SEA-01	Mechanisms exist to facilitate the implementation of industry-recognized security, compliance and resilience practices in the specification, design, development, implementation and modification of Technology Assets, Applications and/or Services (TAAS).	5	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Centralized Management of Security, Compliance & Resilience Controls	SEA-01.1	Mechanisms exist to centrally-manage the organization-wide management and implementation of security, compliance and resilience controls and related processes.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Defense-In-Depth (DiD) Architecture	SEA-03	Mechanisms exist to implement security functions as a layered structure minimizing interactions between layers of the design and avoiding any dependence by lower layers on the functionality or correctness of higher layers.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Application Partitioning	SEA-03.2	Mechanisms exist to separate user functionality from system management functionality.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Process Isolation	SEA-04	Mechanisms exist to implement a separate execution domain for each executing process.	3	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Security Function Isolation	SEA-04.1	Mechanisms exist to isolate security functions from non-security functions.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Hardware Separation	SEA-04.2	Mechanisms exist to implement underlying hardware separation mechanisms to facilitate process separation.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Thread Separation	SEA-04.3	Mechanisms exist to maintain a separate execution domain for each thread in multi-threaded processing.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Information In Shared Resources	SEA-05	Mechanisms exist to prevent unauthorized and unintended information transfer via shared system resources.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Prevent Program Execution	SEA-06	Automated mechanisms exist to prevent the execution of unauthorized software programs.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Fail Secure	SEA-07.2	Mechanisms exist to enable systems to fail to an organization-defined known-state for types of failures, preserving system state information in failure.	3	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Non-Persistence	SEA-08	Mechanisms exist to implement non-persistent system components and services that are initiated in a known state and terminated upon the end of the session of use or periodically at an organization-defined frequency.	3	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	8	
A01:2025	Broken Access Control	Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the user's limits. Common access control vulnerabilities include: Violation of the principle of least privilege, commonly known as deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool that modifies API requests. Permitting viewing or editing someone else's account by providing its unique identifier (insecure direct object references) An accessible API with missing access controls for POST, PUT, and DELETE. Elevation of privilege. Acting as a user without being logged in or or gaining privileges beyond those expected of the logged in user (e.g. admin access). Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation. CORS misconfiguration allows API access from unauthorized or untrusted origins. Force browsing (guessing URLs) to authenticated pages as an unauthenticated user or to privileged pages as a standard user.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	File Integrity Monitoring (FIM)	MON-01.7	Mechanisms exist to utilize a File Integrity Monitor (FIM), or similar change-detection technology, on critical Technology Assets, Applications and/or Services (TAAS) to generate alerts for unauthorized modifications.	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Use of Cryptographic Controls	CRY-01	Mechanisms exist to facilitate the implementation of cryptographic protections controls using known public standards and trusted cryptographic technologies.	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Embedded Technology Security Program	EMB-01	Mechanisms exist to facilitate the implementation of embedded technology controls.	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Data Privacy Requirements for Contractors & Service Providers	PRI-07.1	Mechanisms exist to include data privacy requirements in contracts and other acquisition-related documents that establish data privacy roles and responsibilities for contractors and service providers.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Third-Party Management	TPM-01	Mechanisms exist to facilitate the implementation of third-party management controls.	8	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Supply Chain Risk Management (SCRM)	TPM-03	Mechanisms exist to:(1) Evaluate security risks and threats associated with Technology Assets, Applications and/or Services (TAAS) supply chains; and(2) Take appropriate remediation actions to minimize the organization's exposure to those risks and threats, as necessary.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Third-Party Services	TPM-04	Mechanisms exist to mitigate the risks associated with third-party access to the organization's Technology Assets, Applications, Services and/or Data (TAASD).	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Third-Party Risk Assessments & Approvals	TPM-04.1	Mechanisms exist to conduct a risk assessment prior to the acquisition or outsourcing of technology-related Technology Assets, Applications and/or Services (TAAS).	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	External Connectivity Requirements - Identification of Ports, Protocols & Services	TPM-04.2	Mechanisms exist to require External Service Providers (ESPs) to identify and document the business need for ports, protocols and other services it requires to operate its Technology Assets, Applications and/or Services (TAAS).	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Third-Party Processing, Storage and Service Locations	TPM-04.4	Mechanisms exist to restrict the location of information processing/storage based on business requirements.	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Review of Third-Party Services	TPM-08	Mechanisms exist to monitor, regularly review and assess External Service Providers (ESPs) for compliance with established contractual requirements for security, compliance and resilience controls.	5	
A02:2025	Security Misconfiguration	Security misconfiguration is when a system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities. The application might be vulnerable if: It is missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, testing frameworks, or privileges). Default accounts and their passwords are still enabled and unchanged. A lack of central configuration for intercepting excessive error messages. Error handling reveals stack traces or other overly informative error messages to users. For upgraded systems, the latest security features are disabled or not configured securely. Excessive prioritization of backward compatibility leading to insecure configuration. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values. The server does not send security headers or directives, or they are not set to secure values. Without a concerted, repeatable application security configuration hardening process, systems are at a higher risk.	Functional	Intersects With	Third-Party Deficiency Remediation	TPM-09	Mechanisms exist to address weaknesses or deficiencies in supply chain elements identified during independent or organizational assessments of such elements.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Supply Chain Risk Management (SCRM) Plan	RSK-09	Mechanisms exist to develop a plan for Supply Chain Risk Management (SCRM) associated with the development, acquisition, maintenance and disposal of Technology Assets, Applications and/or Services (TAAS), including documenting selected mitigating actions and monitoring performance against those plans.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Supply Chain Risk Assessment	RSK-09.1	Mechanisms exist to periodically assess supply chain risks associated with Technology Assets, Applications and/or Services (TAAS).	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Software Bill of Materials (SBOM)	TDA-04.2	Mechanisms exist to generate, or obtain, a Software Bill of Materials (SBOM) for Technology Assets, Applications and/or Services (TAAS) that lists software packages in use, including versions and applicable licenses.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Malformed Input Testing	TDA-09.4	Mechanisms exist to utilize testing methods to ensure Technology Assets, Applications and/or Services (TAAS) continue to operate as intended when subject to invalid or unexpected inputs on its interfaces.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Third-Party Management	TPM-01	Mechanisms exist to facilitate the implementation of third-party management controls.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Supply Chain Risk Management (SCRM)	TPM-03	Mechanisms exist to:(1) Evaluate security risks and threats associated with Technology Assets, Applications and/or Services (TAAS) supply chains; and(2) Take appropriate remediation actions to minimize the organization's exposure to those risks and threats, as necessary.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Acquisition Strategies, Tools & Methods	TPM-03.1	Mechanisms exist to utilize tailored acquisition strategies, contract tools and procurement methods for the purchase of unique Technology Assets, Applications and/or Services (TAAS).	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Limit Potential Harm	TPM-03.2	Mechanisms exist to utilize security safeguards to limit harm from potential adversaries who identify and target the organization's supply chain.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Processes To Address Weaknesses or Deficiencies	TPM-03.3	Mechanisms exist to address identified weaknesses or deficiencies in the security of the supply chain	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Third-Party Services	TPM-04	Mechanisms exist to mitigate the risks associated with third-party access to the organization's Technology Assets, Applications, Services and/or Data (TAASD).	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Third-Party Risk Assessments & Approvals	TPM-04.1	Mechanisms exist to conduct a risk assessment prior to the acquisition or outsourcing of technology-related Technology Assets, Applications and/or Services (TAAS).	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	External Connectivity Requirements - Identification of Ports, Protocols & Services	TPM-04.2	Mechanisms exist to require External Service Providers (ESPs) to identify and document the business need for ports, protocols and other services it requires to operate its Technology Assets, Applications and/or Services (TAAS).	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Conflict of Interests	TPM-04.3	Mechanisms exist to ensure that the interests of external service providers are consistent with and reflect organizational interests.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Third-Party Processing, Storage and Service Locations	TPM-04.4	Mechanisms exist to restrict the location of information processing/storage based on business requirements.	5	
A03:2025	Software Supply Chain Failures	Software supply chain failures are breakdowns or other compromises in the process of building, distributing, or updating software. They are often caused by vulnerabilities or malicious changes in third-party code, tools, or other dependencies that the system relies on. You are likely vulnerable if: you do not carefully track the versions of all components that you use (both client-side and server-side). This includes components you directly use as well as nested (transitive) dependencies. the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries. you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use. you do not have a change management process or tracking of changes within your supply chain, including tracking IDEs, IDE extensions and updates, changes to your organization's code repository, sandboxes, image and library repositories, the way artifacts are created and stored, etc. Every part of your supply chain should be documented, especially changes. you have not hardened every part of your supply chain, with a special focus on access control and the application of least privilege. your supply chain systems do not have any separation of duty. No single person should be able to write code and promote it all the way to production without oversight from another human being. components from untrusted sources, across any part of the tech stack, are used in or can impact on production environments. you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure before fixing vulnerabilities. software developers do not test the compatibility of updated, upgraded, or patched libraries. you do not secure the configurations of every part of your system (see A02:2025-Security Misconfiguration). your CI/CD pipeline has weaker security than the systems it builds and deploys,	Functional	Intersects With	Third-Party Contract Requirements	TPM-05	Mechanisms exist to require contractual requirements for applicable security, compliance and resilience requirements with third-parties, reflecting the organization's needs to protect its Technology Assets, Applications, Services and/or Data (TAASD).	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Use of Cryptographic Controls	CRY-01	Mechanisms exist to facilitate the implementation of cryptographic protections controls using known public standards and trusted cryptographic technologies.	8	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Transmission Confidentiality	CRY-03	Cryptographic mechanisms exist to protect the confidentiality of data being transmitted.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Transmission Integrity	CRY-04	Cryptographic mechanisms exist to protect the integrity of data being transmitted.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Encrypting Data At Rest	CRY-05	Cryptographic mechanisms exist to prevent unauthorized disclosure of data at rest.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Public Key Infrastructure (PKI)	CRY-08	Mechanisms exist to securely implement an internal Public Key Infrastructure (PKI) infrastructure or obtain PKI services from a reputable PKI service provider.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Cryptographic Key Management	CRY-09	Mechanisms exist to facilitate cryptographic key management controls to protect the confidentiality, integrity and availability of keys.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Development Methods, Techniques & Processes	TDA-02.3	Mechanisms exist to require software developers to ensure that their software development processes employ industry-recognized secure practices for secure programming, engineering methods, quality control processes and validation techniques to minimize flawed and/or malformed software.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Developer Architecture & Design	TDA-05	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to produce a design specification and security architecture that:(1) Is consistent with and supportive of the organization's security architecture which is established within and is an integrated part of the organization's enterprise architecture;(2) Accurately and completely describes the required security functionality and the allocation of security, compliance and resilience controls among physical and logical components; and(3) Expresses how individual security functions, mechanisms and services work together to provide required security capabilities and a unified approach to protection.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Threat Modeling	TDA-06.2	Mechanisms exist to perform threat modelling and other secure design techniques, to ensure that threats to software and solutions are identified and accounted for.	8	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Software Assurance Maturity Model (SAMM)	TDA-06.3	Mechanisms exist to utilize a Software Assurance Maturity Model (SAMM) to govern a secure development lifecycle for the development of Technology Assets, Applications and/or Services (TAAS).	8	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A04:2025	Cryptographic Failures	Generally speaking, all data in transit should be encrypted at the transport layer (OSI layer 4). Previous hurdles such as CPU performance and private key/certificate management are now handled by CPUs having instructions designed to accelerate encryption (eg: AES support) and private key and certificate management being simplified by services like LetsEncrypt.org with major cloud vendors providing even more tightly integrated certificate management services for their specific platforms. Beyond securing the transport layer, it is important to determine what data needs encryption at rest as well as what data needs extra encryption in transit (at the application layer, OSI layer 7). For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, especially if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations such as PCI Data Security Standard (PCI DSS). For all such data: Are any old or weak cryptographic algorithms or protocols used either by default or in older code? Are default crypto keys in use, are weak crypto keys generated, are keys re-used, or is proper key management and rotation missing? Are crypto keys checked into source code repositories? Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing? Is the received server certificate and the trust chain properly validated? Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate? Are passwords being used as cryptographic keys in the absence of a password based key derivation function? Is randomness used that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability? Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Configuration Management Program	CFG-01	Mechanisms exist to facilitate the implementation of configuration management controls.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Least Functionality	CFG-03	Mechanisms exist to configure systems to provide only essential capabilities by specifically prohibiting or restricting the use of ports, protocols, and/or services.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	File Integrity Monitoring (FIM)	MON-01.7	Mechanisms exist to utilize a File Integrity Monitor (FIM), or similar change-detection technology, on critical Technology Assets, Applications and/or Services (TAAS) to generate alerts for unauthorized modifications.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Embedded Technology Security Program	EMB-01	Mechanisms exist to facilitate the implementation of embedded technology controls.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Embedded Technology Configuration Monitoring	EMB-05	Mechanisms exist to generate log entries on embedded devices when configuration changes or attempts to access interfaces are detected.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Prevent Alterations	EMB-06	Mechanisms exist to protect embedded devices by preventing the unauthorized installation and execution of software.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Restrict Access To Security Functions	END-16	Mechanisms exist to ensure security functions are restricted to authorized individuals and enforce least privilege control requirements for necessary job functions.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Data Privacy Requirements for Contractors & Service Providers	PRI-07.1	Mechanisms exist to include data privacy requirements in contracts and other acquisition-related documents that establish data privacy roles and responsibilities for contractors and service providers.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Secure Engineering Principles	SEA-01	Mechanisms exist to facilitate the implementation of industry-recognized security, compliance and resilience practices in the specification, design, development, implementation and modification of Technology Assets, Applications and/or Services (TAAS).	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Centralized Management of Security, Compliance & Resilience Controls	SEA-01.1	Mechanisms exist to centrally-manage the organization-wide management and implementation of security, compliance and resilience controls and related processes.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Defense-In-Depth (DiD) Architecture	SEA-03	Mechanisms exist to implement security functions as a layered structure minimizing interactions between layers of the design and avoiding any dependence by lower layers on the functionality or correctness of higher layers.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Application Partitioning	SEA-03.2	Mechanisms exist to separate user functionality from system management functionality.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Process Isolation	SEA-04	Mechanisms exist to implement a separate execution domain for each executing process.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Security Function Isolation	SEA-04.1	Mechanisms exist to isolate security functions from non-security functions.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Hardware Separation	SEA-04.2	Mechanisms exist to implement underlying hardware separation mechanisms to facilitate process separation.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Thread Separation	SEA-04.3	Mechanisms exist to maintain a separate execution domain for each thread in multi-threaded processing.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Information In Shared Resources	SEA-05	Mechanisms exist to prevent unauthorized and unintended information transfer via shared system resources.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Prevent Program Execution	SEA-06	Automated mechanisms exist to prevent the execution of unauthorized software programs.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Fail Secure	SEA-07.2	Mechanisms exist to enable systems to fail to an organization-defined known-state for types of failures, preserving system state information in failure.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Non-Persistence	SEA-08	Mechanisms exist to implement non-persistent system components and services that are initiated in a known state and terminated upon the end of the session of use or periodically at an organization-defined frequency.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Third-Party Management	TPM-01	Mechanisms exist to facilitate the implementation of third-party management controls.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Supply Chain Risk Management (SCRM)	TPM-03	Mechanisms exist to:(1) Evaluate security risks and threats associated with Technology Assets, Applications and/or Services (TAAS) supply chains; and(2) Take appropriate remediation actions to minimize the organization's exposure to those risks and threats, as necessary.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Third-Party Services	TPM-04	Mechanisms exist to mitigate the risks associated with third-party access to the organization's Technology Assets, Applications, Services and/or Data (TAASD).	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Third-Party Risk Assessments & Approvals	TPM-04.1	Mechanisms exist to conduct a risk assessment prior to the acquisition or outsourcing of technology-related Technology Assets, Applications and/or Services (TAAS).	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	External Connectivity Requirements - Identification of Ports, Protocols & Services	TPM-04.2	Mechanisms exist to require External Service Providers (ESPs) to identify and document the business need for ports, protocols and other services it requires to operate its Technology Assets, Applications and/or Services (TAAS).	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Third-Party Processing, Storage and Service Locations	TPM-04.4	Mechanisms exist to restrict the location of information processing/storage based on business requirements.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Review of Third-Party Services	TPM-08	Mechanisms exist to monitor, regularly review and assess External Service Providers (ESPs) for compliance with established contractual requirements for security, compliance and resilience controls.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Third-Party Deficiency Remediation	TPM-09	Mechanisms exist to address weaknesses or deficiencies in supply chain elements identified during independent or organizational assessments of such elements.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Vulnerability & Patch Management Program (VPMP)	VPM-01	Mechanisms exist to facilitate the implementation and monitoring of vulnerability management controls.	8	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Vulnerability Remediation Process	VPM-02	Mechanisms exist to ensure that vulnerabilities are properly identified, tracked and remediated.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Vulnerability Ranking	VPM-03	Mechanisms exist to identify and assign a risk ranking to newly discovered security vulnerabilities using reputable outside sources for security vulnerability information.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Continuous Vulnerability Remediation Activities	VPM-04	Mechanisms exist to address new threats and vulnerabilities on an ongoing basis and ensure assets are protected against known attacks.	5	
A05:2025	Injection	An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g. a browser, database, the command line) and causes the interpreter to execute parts of that input as commands. An application is vulnerable to attack when: User-supplied data is not validated, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter. Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records. Potentially hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures. Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Detection is best achieved by a combination of source code review along with automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. The addition of static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can also be helpful to identify injection flaws before production deployment. A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.	Functional	Intersects With	Vulnerability Scanning	VPM-06	Mechanisms exist to detect vulnerabilities and configuration errors by routine vulnerability scanning of systems and applications.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Security, Compliance & Resilience In Project Management	PRM-04	Mechanisms exist to assess security, compliance and resilience controls in system project development to determine the extent to which the controls are implemented correctly, operating as intended and producing the desired outcome with respect to meeting the requirements.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Security, Compliance & Resilience Requirements Definition	PRM-05	Mechanisms exist to identify critical system components and functions by performing a criticality analysis for critical Technology Assets, Applications and/or Services (TAAS) at pre-defined decision points in the Secure Development Life Cycle (SDLC).	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Business Process Definition	PRM-06	Mechanisms exist to define business processes with consideration for security, compliance and resilience that determines:(1) The resulting risk to organizational operations, assets, individuals and other organizations; and(2) Information protection needs arising from the defined business processes and revises the processes as necessary, until an achievable set of protection needs is obtained.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Secure Development Life Cycle (SDLC) Management	PRM-07	Mechanisms exist to ensure changes to Technology Assets, Applications and/or Services (TAAS) within the Secure Development Life Cycle (SDLC) are controlled through formal change control procedures.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS; (2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A06:2025	Insecure Design	Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top Ten risk categories. Note that there is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes, take place at different times in the development process, and have different remediations. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as needed security controls were never created to defend against specific attacks. One of the factors that contributes to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Three key parts of having a secure design are: Gathering Requirements and Resource Management Creating a Secure Design Having a Secure Development Lifecycle	Functional	Intersects With	Unsupported Technology Assets, Applications and/or Services (TAAS)	TDA-17	Mechanisms exist to prevent unsupported Technology Assets, Applications and/or Services (TAAS) by:(1) Removing and/or replacing TAAS when support for the components is no longer available from the developer, vendor or manufacturer; and(2) Requiring justification and documented approval for the continued use of unsupported TAAS required to satisfy mission/business needs.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Identity & Access Management (IAM)	IAC-01	Mechanisms exist to facilitate the implementation of identification and access management controls.	8	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Group Authentication	IAC-02.1	Mechanisms exist to require individuals to be authenticated with an individual authenticator when a group authenticator is utilized.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Replay-Resistant Authentication	IAC-02.2	Automated mechanisms exist to employ replay-resistant authentication.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Acceptance of PIV Credentials	IAC-02.3	Mechanisms exist to accept and electronically verify organizational Personal Identity Verification (PIV) credentials.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Identification & Authentication for Non-Organizational Users	IAC-03	Mechanisms exist to uniquely identify and centrally Authenticate, Authorize and Audit (AAA) third-party users and processes that provide services to the organization.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Acceptance of PIV Credentials from Other Organizations	IAC-03.1	Mechanisms exist to accept and electronically verify Personal Identity Verification (PIV) credentials from third-parties.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Acceptance of Third-Party Credentials	IAC-03.2	Automated mechanisms exist to accept Federal Identity, Credential and Access Management (FICAM)-approved third-party credentials.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Multi-Factor Authentication (MFA)	IAC-06	Automated mechanisms exist to enforce Multi-Factor Authentication (MFA) for:(1) Remote network access;(2) Third-party Technology Assets, Applications and/or Services (TAAS); and/ or(3) Non-console access to critical TAAS that store, transmit and/or process sensitive/regulated data.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Network Access to Privileged Accounts	IAC-06.1	Mechanisms exist to utilize Multi-Factor Authentication (MFA) to authenticate network access for privileged accounts.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Network Access to Non-Privileged Accounts	IAC-06.2	Mechanisms exist to utilize Multi-Factor Authentication (MFA) to authenticate network access for non-privileged accounts.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Local Access to Privileged Accounts	IAC-06.3	Mechanisms exist to utilize Multi-Factor Authentication (MFA) to authenticate local access for privileged accounts.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Identifier Management (User Names)	IAC-09	Mechanisms exist to govern naming standards for usernames and Technology Assets, Applications and/or Services (TAAS).	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Password-Based Authentication	IAC-10.1	Mechanisms exist to enforce complexity, length and lifespan considerations to ensure strong criteria for password-based authentication.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	PKI-Based Authentication	IAC-10.2	Automated mechanisms exist to validate certificates by constructing and verifying a certification path to an accepted trust anchor including checking certificate status information for PKI-based authentication.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Automated Support For Password Strength	IAC-10.4	Automated mechanisms exist to determine if password authenticators are sufficiently strong enough to satisfy organization-defined password length and complexity requirements.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	No Embedded Unencrypted Static Authenticators	IAC-10.6	Mechanisms exist to ensure that unencrypted, static authenticators are not embedded in applications, scripts or stored on function keys.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Default Authenticators	IAC-10.8	Mechanisms exist to ensure default authenticators are changed as part of account creation or system installation.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Re-Authentication	IAC-14	Mechanisms exist to force users and devices to re-authenticate according to organization-defined circumstances that necessitate re-authentication.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Data Privacy Requirements for Contractors & Service Providers	PRI-07.1	Mechanisms exist to include data privacy requirements in contracts and other acquisition-related documents that establish data privacy roles and responsibilities for contractors and service providers.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A07:2025	Authentication Failures	When an attacker is able to trick a system into recognizing an invalid or incorrect user as legitimate, this vulnerability is present. There may be authentication weaknesses if the application: Permits automated attacks such as credential stuffing, where the attacker has a breached list of valid usernames and passwords. More recently this type of attack has been expanded to include hybrid password attacks credential stuffing (also known as password spray attacks), where the attacker uses variations or increments of spilled credentials to gain access, for instance trying Password1!, Password2!, Password3! and so on. Permits brute force or other automated, scripted attacks that are not quickly blocked. Permits default, weak, or well-known passwords, such as "Password1" or "admin" username with an "admin" password. Allows users to create new accounts with already known-breached credentials. Allows use of weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe. Uses plain text, encrypted, or weakly hashed passwords data stores (see A04:2025-Cryptographic Failures). Has missing or ineffective multi-factor authentication. Allows use of weak or ineffective fallbacks if multi-factor authentication is not available. Exposes session identifier in the URL, a hidden field, or another insecure location that is accessible to the client. Reuses the same session identifier after successful login. Does not correctly invalidate user sessions or authentication tokens (mainly single sign-on (SSO) tokens) during logout or a period of inactivity. Does not correctly assert the scope and intended audience of the provided credentials.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Integrity Mechanisms for Software / Firmware Updates	TDA-01.2	Mechanisms exist to utilize integrity validation mechanisms for security updates.	8	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Threat Modeling	TDA-06.2	Mechanisms exist to perform threat modeling and other secure design techniques, to ensure that threats to software and solutions are identified and accounted for.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Software Assurance Maturity Model (SAMM)	TDA-06.3	Mechanisms exist to utilize a Software Assurance Maturity Model (SAMM) to govern a secure development lifecycle for the development of Technology Assets, Applications and/or Services (TAAS).	8	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Input Data Validation	TDA-18	Mechanisms exist to check the validity of information inputs.	5	
A08:2025	Software or Data Integrity Failures	Software and data integrity failures relate to code and infrastructure that does not protect against invalid or untrusted code or data being treated as trusted and valid. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline without consuming and providing software integrity checks can introduce the potential for unauthorized access, insecure or malicious code, or system compromise. Another example of this is a CI/CD that pulls code or artifacts from untrusted places and/or doesn't verify them before use (by checking the signature or similar mechanism). Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.	Functional	Intersects With	Error Handling	TDA-19	Mechanisms exist to handle error conditions by:(1) Identifying potentially security-relevant error conditions;(2) Generating error messages that provide information necessary for corrective actions without revealing sensitive or potentially harmful information in error logs and administrative messages that could be exploited; and(3) Revealing error messages only to authorized personnel.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Continuous Monitoring	MON-01	Mechanisms exist to facilitate the implementation of enterprise-wide monitoring controls.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Intrusion Detection & Prevention Systems (IDS & IPS)	MON-01.1	Mechanisms exist to implement Intrusion Detection / Prevention Systems (IDS / IPS) technologies on critical systems, key network segments and network choke points.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Automated Tools for Real-Time Analysis	MON-01.2	Mechanisms exist to utilize a Security Incident Event Manager (SIEM), or similar automated tool, to support near real-time analysis and incident escalation.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Inbound & Outbound Communications Traffic	MON-01.3	Mechanisms exist to continuously monitor inbound and outbound communications traffic for unusual or unauthorized activities or conditions.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	System Generated Alerts	MON-01.4	Mechanisms exist to generate, monitor, correlate and respond to alerts from physical, cybersecurity, data protection and supply chain activities to achieve integrated situational awareness.	8	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	File Integrity Monitoring (FIM)	MON-01.7	Mechanisms exist to utilize a File Integrity Monitor (FIM), or similar change-detection technology, on critical Technology Assets, Applications and/or Services (TAAS) to generate alerts for unauthorized modifications.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Security Event Monitoring	MON-01.8	Mechanisms exist to review event logs on an ongoing basis and escalate incidents in accordance with established timelines and procedures.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Deactivated Account Activity	MON-01.10	Mechanisms exist to monitor deactivated accounts for attempted usage.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Analyze and Prioritize Monitoring Requirements	MON-01.16	Mechanisms exist to assess the organization's needs for monitoring and prioritize the monitoring of Technology Assets, Applications and/or Services (TAAS), based on TAAS criticality and the sensitivity of the data it stores, transmits and processes.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Centralized Collection of Security Event Logs	MON-02	Mechanisms exist to utilize a Security Incident Event Manager (SIEM), or similar automated tool, to support the centralized collection of security-related event logs.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Correlate Monitoring Information	MON-02.1	Automated mechanisms exist to correlate both technical and non-technical information from across the enterprise by a Security Incident Event Manager (SIEM) or similar automated tool, to enhance organization-wide situational awareness.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Content of Event Logs	MON-03	Mechanisms exist to configure Technology Assets, Applications and/or Services (TAAS) to produce event logs that contain sufficient information to, at a minimum:(1) Establish what type of event occurred;(2) When (date and time) the event occurred;(3) Where the event occurred;(4) The source of the event;(5) The outcome (success or failure) of the event; and(6) The identity of any user/subject associated with the event.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Audit Trails	MON-03.2	Mechanisms exist to link system access to individual users or service accounts.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Privileged Functions Logging	MON-03.3	Mechanisms exist to log and review the actions of users and/or services with elevated privileges.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Verbosity Logging for Boundary Devices	MON-03.4	Mechanisms exist to verbosely log all traffic (both allowed and blocked) arriving at network boundary devices, including firewalls, Intrusion Detection / Prevention Systems (IDS/IPS) and inbound and outbound proxies.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Response To Event Log Processing Failures	MON-05	Mechanisms exist to alert appropriate personnel in the event of a log processing failure and take actions to remedy the disruption.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Query Parameter Audits of Personal Data (PD)	MON-06.1	Mechanisms exist to provide and implement the capability for auditing the parameters of user query events for data sets containing Personal Data (PD).	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Time Stamps	MON-07	Mechanisms exist to configure Technology Assets, Applications and/or Services (TAAS) to use an authoritative time source to generate time stamps for event logs.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Synchronization With Authoritative Time Source	MON-07.1	Mechanisms exist to synchronize internal system clocks with an authoritative time source.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Protection of Event Logs	MON-08	Mechanisms exist to protect event logs and audit tools from unauthorized access, modification and deletion.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Event Log Backup on Separate Physical Systems / Components	MON-08.1	Mechanisms exist to back up event logs onto a physically different system or system component than the Security Incident Event Manager (SIEM) or similar automated tool.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Access by Subset of Privileged Users	MON-08.2	Mechanisms exist to restrict access to the management of event logs to privileged users with a specific business need.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Event Log Retention	MON-10	Mechanisms exist to retain event logs for a time period consistent with records retention requirements to provide support for after-the-fact investigations of security incidents and to meet statutory, regulatory and contractual retention requirements.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Embedded Technology Security Program	EMB-01	Mechanisms exist to facilitate the implementation of embedded technology controls.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Data Privacy Requirements for Contractors & Service Providers	PRI-07.1	Mechanisms exist to include data privacy requirements in contracts and other acquisition-related documents that establish data privacy roles and responsibilities for contractors and service providers.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A09:2025	Security Logging & Alerting Failures	Without logging and monitoring, attacks and breaches cannot be detected, and without alerting it is very difficult to respond quickly and effectively during a security incident. Insufficient logging, continuous monitoring, detection, and alerting to initiate active responses occurs any time: Auditable events, such as logins, failed logins, and high-value transactions, are not logged or logged inconsistently (for instance, only logging successful logins, but not failed attempts). Warnings and errors generate no, inadequate, or unclear log messages. The integrity of logs is not properly protected from tampering. Logs of applications and APIs are not monitored for suspicious activity. Logs are only stored locally, and not properly backedup. Appropriate alerting thresholds and response escalation processes are not in place or effective. Alerts are not received or reviewed within a reasonable amount of time. Penetration testing and scans by dynamic application security testing (DAST) tools (such as Burp or ZAP) do not trigger alerts. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time. You are vulnerable to sensitive information leakage by making logging and alerting events visible to a user or an attacker (see A01:2025-Broken Access Control), or by logging sensitive information that should not be logged (such as PII or PHI). You are vulnerable to injections or attacks on the logging or monitoring systems if log data is not correctly encoded. The application is missing or mishandling errors and other exceptional conditions, such that the system is unaware there was an error, and is therefore unable to log there was a problem. Adequate 'use cases' for issuing alerts are missing or outdated to recognize a special situation. Too many false positive alerts make it impossible to distinguish important alerts from unimportant ones, resulting in them being recognized too late or not at all (physical overload of the SOC team). Detected alerts cannot be processed correctly because the playbook for the use case is incomplete, out of date, or missing.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Secure Baseline Configurations	CFG-02	Mechanisms exist to develop, document and maintain secure baseline configurations for Technology Assets, Applications and/or Services (TAAS) that are consistent with industry-accepted system hardening standards.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Embedded Technology Security Program	EMB-01	Mechanisms exist to facilitate the implementation of embedded technology controls.	8	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Specialized Assessments	IAO-02.2	Mechanisms exist to conduct specialized assessments for:(1) Statutory, regulatory and contractual compliance obligations;(2) Monitoring capabilities;(3) Mobile devices;(4) Databases;(5) Application security;(6) Embedded technologies (e.g., IoT, OT, etc.);(7) Vulnerability management;(8) Malicious code;(9) Insider threats;(10) Performance/load testing; and/or(11) Artificial Intelligence and Autonomous Technologies (AAT).	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Threat Analysis & Flaw Remediation During Development	IAO-04	Mechanisms exist to require system developers and integrators to create and execute a Security Testing and Evaluation (ST&E) plan, or similar process, to identify and remediate flaws during development.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Data Privacy Requirements for Contractors & Service Providers	PRI-07.1	Mechanisms exist to include data privacy requirements in contracts and other acquisition-related documents that establish data privacy roles and responsibilities for contractors and service providers.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Technology Development & Acquisition	TDA-01	Mechanisms exist to facilitate the implementation of tailored development and acquisition strategies, contract tools and procurement methods to meet unique business needs.	8	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Product Management	TDA-01.1	Mechanisms exist to design and implement product management processes to proactively govern the design, development and production of Technology Assets, Applications and/or Services (TAAS) across the System Development Life Cycle (SDLC) to:(1) Improve functionality;(2) Enhance security and resiliency capabilities;(3) Correct security deficiencies; and(4) Conform with applicable statutory, regulatory and/or contractual obligations.	8	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Minimum Viable Product (MVP) Security Requirements	TDA-02	Mechanisms exist to design, develop and produce Technology Assets, Applications and/or Services (TAAS) in such a way that risk-based technical and functional specifications ensure Minimum Viable Product (MVP) criteria establish an appropriate level of security and resiliency based on applicable risks and threats.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Documentation Requirements	TDA-04	Mechanisms exist to obtain, protect and distribute administrator documentation for Technology Assets, Applications and/or Services (TAAS) that describe:(1) Secure configuration, installation and operation of the TAAS;(2) Effective use and maintenance of security features/functions; and(3) Known vulnerabilities regarding configuration and use of administrative (e.g., privileged) functions.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Functional Properties	TDA-04.1	Mechanisms exist to require software developers to provide information describing the functional properties of the security, compliance and resilience controls to be utilized within Technology Assets, Applications and/or Services (TAAS) in sufficient detail to permit analysis and testing of the controls.	5	

FDE #	FDE Name	Focal Document Element (FDE) Description	STRM Rationale	STRM Relationship	SCF Control	SCF #	Secure Controls Framework (SCF) Control Description	Strength of Relationship	Notes
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Secure Software Development Practices (SSDP)	TDA-06	Mechanisms exist to develop applications based on Secure Software Development Practices (SSDP).	8	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Security, Compliance & Resilience Testing Throughout Development	TDA-09	Mechanisms exist to require system developers/integrators consult with security, compliance and/or resilience personnel to:(1) Create and implement a Security Testing and Evaluation (ST&E) plan, or similar capability;(2) Implement a verifiable flaw remediation process to correct weaknesses and deficiencies identified during the control testing and evaluation process; and(3) Document the results.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Static Code Analysis	TDA-09.2	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ static code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Dynamic Code Analysis	TDA-09.3	Mechanisms exist to require the developers of Technology Assets, Applications and/or Services (TAAS) to employ dynamic code analysis tools to identify and remediate common flaws and document the results of the analysis.	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Application Penetration Testing	TDA-09.5	Mechanisms exist to perform application-level penetration testing of custom-made Technology Assets, Applications and/or Services (TAAS).	5	
A10:2025	Mishandling of Exceptional Conditions	Mishandling exceptional conditions in software happens when programs fail to prevent, detect, and respond to unusual and unpredictable situations, which leads to crashes, unexpected behavior, and sometimes vulnerabilities. This can involve one or more of the following 3 failings; the application doesn't prevent an unusual situation from happening, it doesn't identify the situation as it is happening, and/or it responds poorly or not at all to the situation afterwards. Exceptional conditions can be caused by missing, poor, or incomplete input validation, or late, high level error handling instead at the functions where they occur, or unexpected environmental states such as memory, privilege, or network issues, inconsistent exception handling, or exceptions that are not handled at all, allowing the system to fall into an unknown and unpredictable state. Any time an application is unsure of its next instruction, an exceptional condition has been mishandled. Hard-to-find errors and exceptions can threaten the security of the whole application for a long time. Many different security vulnerabilities can happen when we mishandle exceptional conditions, such as logic bugs, overflows, race conditions, fraudulent transactions, or issues with memory, state, resource, timing, authentication, and authorization. These types of vulnerabilities can negatively affect the confidentiality, availability, and/or integrity of a system or it's data. Attackers manipulate an application's flawed error handling to strike this vulnerability.	Functional	Intersects With	Error Handling	TDA-19	Mechanisms exist to handle error conditions by:(1) Identifying potentially security-relevant error conditions;(2) Generating error messages that provide information necessary for corrective actions without revealing sensitive or potentially harmful information in error logs and administrative messages that could be exploited; and(3) Revealing error messages only to authorized personnel.	5	