Top Ten Security Risks

	Risk	Discription	Exploitability	Prevalence	Detectability	Impacts	How to Prevent
1	Injection	Injection flaws occur when untrusted data is sent to an interpreter as part of a command, tricking it into executing unintended commands or accessing data without proper authorization.	Easy	Common	Easy	Severe	Preventing injection requires keeping data separate from commands and queries. • The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs). Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec(). • Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications. • For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter. Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software. • Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection
2	Broken Authentication	Authentication and session management are often implemented incorrectly allowing attackers to assume other users' identities	Easy	Common	Average	Severe	Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks. • Do not ship or deploy with any default credentials, particularly for admin users. • Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords. • Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies. • Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes. • Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected. • Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.
3	Sensitive Data Exposure	Applications and APIs that don't properly protect sensitive data allow attackers to steal it to commit credit card fraud, identity theft or other crimes. Sensitive data requires special precautions when exchanged with the browser.	Average	Widespread	Average	Severe	Do the following, at a minimum, and consult the references: • Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs. • Apply controls as per the classification. • Don’t store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen. • Make sure to encrypt all sensitive data at rest. • Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management. • Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS). • Disable caching for responses that contain sensitive data. • Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2. • Verify independently the effectiveness of configuration and settings.
4	XML External Entities	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks	Average	Common	Easy	Severe	Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires: • Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data. • Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher. • Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'. • Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes. • Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar. • SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations. If these controls are not possible, consider using virtual patching, API security gateways, or Web Application
5	Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users’ data, change access rights, etc.	Average	Common	Average	Severe	Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata. • With the exception of public resources, deny by default. • Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage. • Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record. • Unique application business limit requirements should be enforced by domain models. • Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots. • Log access control failures, alert admins when appropriate (e.g. repeated failures). • Rate limit API and controller access to minimize the harm from automated attack tooling. • JWT tokens should be invalidated on the server after logout. Developers and QA staff s

Top Ten Security Risks

	Risk	Discription	Exploitability	Prevalence	Detectability	Impacts	How to Prevent
6	Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion	Easy	Widespread	Easy	Moderate	Secure installation processes should be implemented, including: • A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to setup a new secure environment. • A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks. • A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process (see A9:2017-Using Components with Known Vulnerabilities). In particular, review cloud storage permissions (e.g. S3 bucket permissions). • A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups. • Sending security directives to clients, e.g. Security Headers. • An automated process to verify the effectiveness of the configurations and settings in all environments.
7	Cross-Site Scripting	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.	Easy	Widespread	Easy	Moderate	Preventing XSS requires separation of untrusted data from active browser content. This can be achieved by: • Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered. • Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP Cheat Sheet 'XSS Prevention' has details on the required data escaping techniques. • Applying context-sensitive encoding when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the OWASP Cheat Sheet 'DOM based XSS Prevention'. • Enabling a Content Security Policy (CSP) is a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks).
8	Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.	Difficult	Common	Average	Severe	The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types. If that is not possible, consider one of more of the following: • Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering. • Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable. • Isolating and running code that deserializes in low privilege environments when possible. • Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions. • Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize. • Monitoring deserialization, alerting if a user deserializes constantly
9	Using Components with Know Vulnerabilites	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.	Average	Widespread	Average	Moderate	There should be a patch management process in place to: • Remove unused dependencies, unnecessary features, components, files, and documentation. • Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck, retire.js, etc. Continuously monitor sources like CVE and NVD for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use. • Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component. • Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue. Every organization must ensure that there is an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.
10	Insufficient Logging and Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.	Average	Widespread	Difficult	Moderate	As per the risk of the data stored or processed by the application: • Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis. • Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions. • Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar. • Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion. • Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later. There are commercial and open source application protection frameworks such as OWASP AppSensor, web application firewalls such as ModSecurity with the OWASP ModSecurity Core Rule Set, and log correlation software with custom dashboards and alerting.