OWASP- Open Web Application Security Project

2017	2021
A01:2017-Injection	A01:2021-Broken Access Control
A02:2017-Broken Authentication	A02:2021-Cryptographic Failures
A03:2017-Sensitive Data Exposure	A03:2021-Injection
A04:2017-XML External Entities(XXE)	A04:2021-Insecure Design
A05:2017-Broken Access Control	A05:2021-Security Misconfiguration
A06:2017-Security Misconfiguration	A06:2021-Vilnerable And Outdated Components
A07:2017-Cross-Site Scripting(XSS)	A07:2021-Identification and Authentication Failuers
A08:2017-Insecure Deserialization	A08:2021-Software And Data Integrity Failures
A09:2017-Using Components With Known Vulnerabilities	A09:2021-Security logging and Monitoring Failures
A10:2017-Insufficient Logging & Monitoring	A10:2021 Server-Side Request Forgery

BugBotIO

A01:2021-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:

- Violation of the principle of least privilege or deny by default, where access should only be granted for 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 modifying API requests.
- Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references)
- Accessing API with missing access controls for POST, PUT and DELETE.
- Elevation of privilege. Acting as a user without being logged in or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.

- CORS misconfiguration allows API access from unauthorized/untrusted origins.
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user.

How To Prevent:

1. Except for public resources, deny by default.
2. Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
3. Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.
4. Unique application business limit requirements should be enforced by domain models.
5. Disable web server directory listing and ensure file metadata (e.g., .git) and backup files are not present within web roots.
6. Log access control failures, alert admins when appropriate (e.g., repeated failures).
7. Rate limit API and controller access to minimize the harm from automated attack tooling.
8. Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.

BugBot19

A02:2021-Cryptographic Failures

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, mainly if that data falls under privacy laws, e.g.EU's General Data Protection Regulation (GDPR), or regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS).

How to prevent

1. Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
2. 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.
3. Make sure to encrypt all sensitive data at rest.

4. Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
5. Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
6. Disable caching for response that contain sensitive data.
7. Apply required security controls as per the data classification.
8. Do not use legacy protocols such as FTP and SMTP for transporting sensitive data.
9. Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt or PBKDF2.
10. Initialization vectors must be chosen appropriate for the mode of operation. For many modes, this means using a CSPRNG (cryptographically secure pseudo random number generator). For modes that require a nonce, then the initialization vector (IV) does not need a CSPRNG. In all cases, the IV should never be used twice for a fixed key.
11. Always use authenticated encryption instead of just encryption.
12. Keys should be generated cryptographically randomly and stored in memory as byte arrays. If a password is used, then it must be converted to a key via an appropriate password base key derivation function.
13. Ensure that cryptographic randomness is used where appropriate, and that it has not been seeded in a predictable way or with low entropy. Most modern APIs do not require the developer to seed the CSPRNG to get security.
14. Avoid deprecated cryptographic functions and padding schemes, such as MD5, SHA1, PKCS number 1 v1.5 .
15. Verify independently the effectiveness of configuration and settings

A03:2021-Injection

An application is vulnerable to attack when:

1. User-supplied data is not validated, filtered, or sanitized by the application.
2. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
3. Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
4. Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

BugBot19

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. Source code review is the best method of detecting if applications are vulnerable to injections. Automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs is strongly encouraged. Organizations can include static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline to identify introduced injection flaws before production deployment.

How to prevent

Preventing injection requires keeping data separate from commands and queries:

1. The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to 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().
2. Use positive 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.
3. For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
   Note: SQL structures 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.
4. Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

A04:2021-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 10 risk categories. 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 and remediation. 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 by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in

BugBot19

the software or system being developed, and thus the failure to determine what level of security design is required.

Requirements and Resource Management

Collect and negotiate the business requirements for an application with the business, including the protection requirements concerning confidentiality, integrity, availability, and authenticity of all data assets and the expected business logic. Take into account how exposed your application will be and if you need segregation of tenants (additionally to access control). Compile the technical requirements, including functional and non-functional security requirements. Plan and negotiate the budget covering all design, build, testing, and operation, including security activities.

How to prevent

1. Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls
2. Establish and use a library of secure design patterns or paved road ready to use components
3. Use threat modeling for critical authentication, access control, business logic, and key flows
4. Integrate security language and controls into user stories
5. Integrate plausibility checks at each tier of your application (from frontend to backend)
6. Write unit and integration tests to validate that all critical flows are resistant to the threat model. Compile use-cases and misuse-cases for each tier of your application.
7. Segregate tier layers on the system and network layers depending on the exposure and protection needs
8. Segregate tenants robustly by design throughout all tiers
9. Limit resource consumption by user or service.

BugBot19

A05:2021-Security Misconfiguration

Description

The application might be vulnerable if the application is:

1. Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
2. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
3. Default accounts and their passwords are still enabled and unchanged.

4. Error handling reveals stack traces or other overly informative error messages to users.
5. For upgraded systems, the latest security features are disabled or not configured securely.
6. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
7. The server does not send security headers or directives, or they are not set to secure values.
8. The software is out of date or vulnerable (see A06:2021-Vulnerable and Outdated Components).

How to prevent

1. Secure installation processes should be implemented, including:
2. A repeatable hardening process makes it fast and easy to deploy another environment that is appropriately 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 set up a new secure environment.
3. A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
4. A task to review and update the configurations appropriate to all security notes, updates, and patches as part of the patch management process (see A06:2021-Vulnerable and Outdated Components). Review cloud storage permissions (e.g., S3 bucket permissions).
5. A segmented application architecture provides effective and secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).
6. Sending security directives to clients, e.g., Security Headers.
7. An automated process to verify the effectiveness of the configurations and settings in all environments.

A06:2021-Vulnerable and Outdated Components

Description

You are likely vulnerable:

1. If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.

BugBot19

2. If 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.
3. If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
4. If 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 to fixed vulnerabilities.
5. If software developers do not test the compatibility of updated, upgraded, or patched libraries.
6. If you do not secure the components' configurations (see A05:2021-Security Misconfiguration).

How to prevent

1. Remove unused dependencies, unnecessary features, components, files, and documentation.
2. Continuously inventory the versions of both client-side and server-side components (e.g., frameworks, libraries) and their dependencies using tools like versions, OWASP Dependency Check, retire.js, etc. Continuously monitor sources like Common Vulnerability and Exposures (CVE) and National Vulnerability Database (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.
3. Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component (See A08:2021-Software and Data Integrity Failures).
4. 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.

A07:2021-Identification and Authentication Failures

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:

BugBot19

1. Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
2. Permits brute force or other automated attacks.
3. Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
4. Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
5. Uses plain text, encrypted, or weakly hashed passwords data stores (see A02:2021-Cryptographic Failures).
6. Has missing or ineffective multi-factor authentication.
7. Exposes session identifier in the URL.
8. Reuse session identifier after successful login.
9. Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

How to Prevent

1. Where possible, implement multi-factor authentication to prevent automated credential stuffing, brute force, and stolen credential reuse attacks.
2. Do not ship or deploy with any default credentials, particularly for admin users.
3. Implement weak password checks, such as testing new or changed passwords against the top 10,000 worst passwords list.
4. Align password length, complexity, and rotation policies with National Institute of Standards and Technology (NIST) 800-63b's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence-based password policies.
5. Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
6. Limit or increasingly delay failed login attempts, but be careful not to create a denial of service scenario. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
7. Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session identifier should not be in the URL, be securely stored, and invalidated after logout, idle, and absolute timeouts.

A08:2021-Software and Data Integrity Failures

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. 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 can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many

Bugbot19

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.

How to prevent

1. Use digital signatures or similar mechanisms to verify the software or data is from the expected source and has not been altered.
2. Ensure libraries and dependencies, such as npm or Maven, are consuming trusted repositories. If you have a higher risk profile, consider hosting an internal known-good repository that's vetted.
3. Ensure that a software supply chain security tool, such as OWASP Dependency Check or OWASP CycloneDX, is used to verify that components do not contain known vulnerabilities
4. Ensure that there is a review process for code and configuration changes to minimize the chance that malicious code or configuration could be introduced into your software pipeline.
5. Ensure that your CI/CD pipeline has proper segregation, configuration, and access control to ensure the integrity of the code flowing through the build and deploy processes.
6. Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data

A09:2021-Security Logging and Monitoring Failures

Description

Returning to the OWASP Top 10 2021, this category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time:

1. Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
2. Warnings and errors generate no, inadequate, or unclear log messages.
3. Logs of applications and APIs are not monitored for suspicious activity.
4. Logs are only stored locally.

BugBot19

5. Appropriate alerting thresholds and response escalation processes are not in place or effective.
6. Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
7. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

How to prevent

Developers should implement some or all the following controls, depending on the risk of the application:

1. Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.
2. Ensure that logs are generated in a format that log management solutions can easily consume.
3. Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
4. Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
5. DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.
6. Establish or adopt an incident response and recovery plan, such as National Institute of Standards and Technology (NIST) 800-61r2 or later.

There are commercial and open-source application protection frameworks such as the OWASP ModSecurity Core Rule Set, and open-source log correlation software, such as the Elasticsearch, Logstash, Kibana (ELK) stack, that feature custom dashboards and alerting.

A10:2021-Server-Side Request Forgery

Description

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).As modern web applications provide end-users with convenient features, fetching a URL becomes a common

BugBot19

scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

How to prevent

From Network layer

i. Segment remote resource access functionality in separate networks to reduce the impact of SSRF
ii. Enforce “deny by default” firewall policies or network access control rules to block all but essential intranet traffic.

From Application layer:

i. Sanitize and validate all client-supplied input data
ii. Enforce the URL schema, port, and destination with a positive allow list
iii. Do not send raw responses to clients
iv. Disable HTTP redirections
v. Be aware of the URL consistency to avoid attacks such as DNS rebinding and “time of check, time of use” (TOCTOU) race conditions

BugBot19