CHECKLIST

Async Try-Catch Security Checklist for JavaScript

A practical checklist for validating secure async try-catch usage in JavaScript before deployment. Use it to confirm catch boundaries, failure classification, logging safety, and production readiness.

Async Try-Catch Security Checklist for JavaScript

Use this checklist to review async error handling before deployment. It is designed for request handlers, job processors, and other production code that uses async/await.

1) Define the failure boundary

- [ ] Identify the exact awaited operation that can fail.
- [ ] Keep the try-catch block as small as possible.
- [ ] Confirm the block wraps only the operations you intend to classify or recover from.
- [ ] Avoid surrounding unrelated validation, transformation, or persistence logic unless it belongs to the same failure boundary.
- [ ] Confirm you are not using try-catch as a generic safety net for the entire function.

2) Verify what is actually awaited

- [ ] Check that every promise that must be handled is preceded by await inside the protected scope.
- [ ] Search for fire-and-forget calls that create promises without awaiting them.

- [ ] Confirm that unawaited promises have their own supervision or error handling path.
- [ ] Verify that promise chains inside the block do not escape the intended catch boundary.
- [ ] Confirm callback-based code inside the block has separate error handling if it can fail asynchronously.

3) Classify failures before responding

- [ ] Distinguish between input errors, authentication failures, dependency timeouts, rate limits, and unexpected exceptions.
- [ ] Return a response that matches the failure type instead of using one generic fallback.
- [ ] Mark retryable failures clearly when the caller or upstream system needs that signal.
- [ ] Fail fast on invalid input rather than retrying it.
- [ ] Re-throw unexpected errors when the current layer cannot safely decide a response.

4) Protect sensitive data in logs

- [ ] Log only non-sensitive context.
- [ ] Include correlation IDs or request IDs where available.
- [ ] Avoid logging secrets, tokens, full request payloads, or account identifiers unless strictly necessary.
- [ ] Confirm error logs do not expose internal stack traces to end users.
- [ ] Review fallback paths to ensure they do not leak security-relevant details.

5) Preserve deterministic control flow

- [ ] Make sure every code path returns, throws, or continues intentionally.
- [ ] Confirm the handler does not hang after a caught rejection.
- [ ] Verify that partial state cannot be mistaken for success.
- [ ] Check that cleanup or rollback logic runs when needed.

- [ ] Ensure the catch block does not hide failures behind a success-shaped response.

6) Avoid overbroad catch blocks

- [ ] Split large blocks into smaller units if they combine validation, remote calls, and persistence.
- [ ] Keep unrelated work outside the protected scope when possible.
- [ ] Use separate try-catch blocks when different failure types require different actions.
- [ ] Confirm that a catch block is not masking the real failing operation.
- [ ] Prefer explicit error handling over silent fallback behavior.

7) Validate realistic failure behavior

- [ ] Test dependency timeouts and connection failures.
- [ ] Test malformed input and missing required fields.
- [ ] Test authentication and authorization failures.
- [ ] Test downstream rejection after a successful upstream call.
- [ ] Test audit, telemetry, or logging failures if they affect the workflow.
- [ ] Confirm the implementation behaves correctly under both synchronous throws and promise rejections.

8) Check response and retry policy

- [ ] Confirm the caller receives the right status or error object.
- [ ] Verify retry logic applies only to retryable conditions.
- [ ] Confirm the system does not retry invalid requests.
- [ ] Avoid automatic retries that can duplicate side effects.
- [ ] Ensure idempotency is considered before retrying operations that write data.

9) Confirm boundary ownership

- [ ] Use try-catch at the layer that can make the safest decision.
- [ ] Let errors bubble upward when the caller owns the response policy.
- [ ] In library code, prefer rejected promises or typed errors over deciding application policy.
- [ ] In request handlers, catch only what you can classify and handle locally.
- [ ] Re-throw errors after adding only safe, useful context.

10) Final pre-deployment review

- [ ] No promise is left unawaited inside a protected scope.
- [ ] No catch block returns a misleading success response.
- [ ] No log entry includes sensitive payload data.
- [ ] No broad catch block hides the real operation failure.
- [ ] Failure paths have been tested with realistic rejection cases.
- [ ] The code can distinguish recoverable failures from unexpected defects.

Quick go-live test

Before deploying, answer these three questions:

- Can this catch block make a safer decision than its caller?
- If no, move it upward.
- Does the catch block cover only awaited work that should be handled here?
- If no, narrow it.
- Have I tested the actual failure modes I expect in production?
- If no, validate before release.

Recommended rule of thumb

Use async try-catch for the smallest meaningful unit of awaited work, classify known failures, log only safe context, and let everything else propagate to the layer with better decision-making authority.