



## SUNIMOD FIELD NOTES

# The Payment You Approved Can Change: How Stale Approvals Break Dual Control

---

An approval attached to an editable request can authorize a different payment, refund, or infrastructure change. Learn how stale approvals work and how immutable instructions, independent review, and transactional release protect your business.

**BY** Christopher Jacobson

**PUBLISHED** October 15, 2026

**TOPIC** CISSP, Domain 2

<https://sunimod.com/stale-approvals-dual-control-workflow-security/>

**The business problem:** Your application remembers that a manager approved a request, but forgets which amount, destination, scope, or configuration the manager approved. The practical fix is to bind approval to an immutable instruction, enforce independent review, and make the execution path use that same instruction.

## In this guide

1. [The business rule that breaks](#)
2. [How stale approval is exploited](#)
3. [Reproduce the defect locally](#)
4. [Build an immutable approval gate](#)
5. [Verify the security properties](#)
6. [Protect the execution boundary](#)
7. [Business repercussions and a practical remediation project](#)

## The business rule that breaks

Imagine a fictitious company with an internal payment portal. One employee prepares a \$250 vendor payment. A manager checks the amount and destination, then approves it. An automated worker processes approved payments later.

The written policy sounds sensible: the person requesting a payment cannot approve it. But suppose the requester can still edit the record after approval. They change the amount to \$25,000 and select a different destination. The worker sees the existing approval and processes the current record.

Two people participated. Only one reviewed the final instruction. The application has preserved the appearance of independent approval while losing the control that made that approval meaningful.

[NIST describes separation of duty](#) as restricting combinations of privileges so one person cannot misuse a system alone. For this workflow, that requires more than assigning different role names. The independent person must approve the same consequential action that the system eventually authorizes for execution.

## Identity is not transaction integrity

A successful login establishes identity according to the authentication mechanism. It does not establish permission for every later action. [OWASP explicitly distinguishes authentication from authorization](#) and recommends checking permissions on every request.

In our example, stronger login protection alone would not repair the flaw. Even with legitimate accounts and successful multifactor authentication, the software could still connect an old approval to changed payment details. The missing relationship is between the approval and the instruction, not merely between the user and the account.

## How stale approval is exploited

The necessary conditions are specific: an actor can influence consequential details after review, the original approval remains usable, and an execution path trusts that approval without establishing that the reviewed details still match. Without those conditions, this particular attack chain does not work.

1. **Prepare an acceptable request.** The actor submits a payment, refund, access grant, or operational change that appears appropriate for approval.
2. **Obtain a legitimate sign-off.** A reviewer sees the acceptable version. The application records an approver or an approved status against the request identifier.
3. **Change a consequential detail.** An edit form, bulk import, integration, or another permitted write path modifies the amount, destination, environment, or scope.
4. **Keep the original approval attached.** The software does not invalidate the sign-off or require a new review.
5. **Execute the current record.** A worker checks that an approval exists, then reads the changed details.
6. **Leave misleading evidence.** An activity screen associates the reviewer with the request, even though the reviewer never approved the action that occurred.

This is a business-logic failure, not necessarily a password attack, SQL injection, or publicly cataloged product vulnerability. [OWASP identifies modifying orders after approval](#) as an example of exploiting workflow gaps.

### The indirect destination trap

A request does not have to change visibly. Suppose the approved record contains only a vendor identifier. The worker later looks up that vendor's current bank destination. An actor changes the vendor record rather than the payment request. The amount and request identifier remain unchanged, but the instruction now resolves to somewhere different.

For this design, the approval must therefore cover a fixed destination or an immutable destination version. A label such as “approved vendor” is not enough when the label points to editable instructions.

### A race is not required

The basic attack can happen through ordinary sequential actions separated by minutes or hours. Some implementations also introduce a timing vulnerability when a concurrent update occurs between a final check and execution. [MITRE's time-of-check/time-of-use description](#) covers that narrower race-condition pattern. Do not limit testing to fast concurrent requests and overlook the slower workflow defect.

## Reproduce the defect locally

**Authorized, local testing only.** These examples use fictitious identities and nonfunctional destination references. They make no network requests and send no payments. The first example is deliberately vulnerable; run it only as an isolated local exercise, not as application code.

Save the following as `stale_approval.py`. It uses a real in-memory SQLite database. The update represents what a permitted edit operation could do behind an application's interface; it is not an instruction to gain direct access to somebody else's database.

#### Python · expose a stale approval with real SQLite queries

```
import sqlite3

with sqlite3.connect(":memory:") as db:
    db.executescript("""
        CREATE TABLE requests (
            id INTEGER PRIMARY KEY,
            amount_minor INTEGER NOT NULL,
            destination TEXT NOT NULL
        );
        CREATE TABLE approvals (
            request_id INTEGER PRIMARY KEY,
            approved_by TEXT NOT NULL
        );
        INSERT INTO requests VALUES (1, 25000, 'destination-original-v1');
        INSERT INTO approvals VALUES (1, 'checker@example.com');

        UPDATE requests
        SET amount_minor = 2500000, destination = 'destination-other-v1'
        WHERE id = 1;
    """)
    row = db.execute("""
        SELECT r.amount_minor, r.destination, a.approved_by
        FROM requests AS r
        JOIN approvals AS a ON a.request_id = r.id
        WHERE r.id = 1
    """).fetchone()
    cents, destination, approver = row
    print(f"Selected amount: USD {cents // 100}.{cents % 100:02d}")
    print(f"Selected destination: {destination}")
    print(f"Approval attributed to: {approver}")
db.close()
```

On Ubuntu with Python installed, run:

#### Bash · run the isolated vulnerable example

```
python3 stale_approval.py
```

The selected execution details are:

#### Plaintext · changed details selected under the original approval

```
Selected amount: USD 25000.00
Selected destination: destination-other-v1
Approval attributed to: checker@example.com
```

The original amount was 25,000 cents, or \$250. The query now selects 2,500,000 cents, or \$25,000, together with a different destination and the original approver's identity.

The defect is in the relationship. The approval identifies a request, but the request's meaning is editable. Joining on the identifier proves that someone approved something associated with that identifier. It does not prove that they approved the values returned now.

No payment is queued or transmitted by this sample. It exposes the wrong data selection that a vulnerable execution worker could rely on.

## Build an immutable approval gate

One repair is to make the submitted instruction immutable. A meaningful change becomes a new request with a new identifier and requires fresh approval. The old request must be cancelled before a replacement proceeds, provided it has not already crossed the release boundary.

[OWASP's transaction-authorization guidance](#) recommends showing significant transaction details to the person authorizing them, protecting those details against modification, enforcing checks on the server, and verifying authorization before execution. The implementation below applies those ideas to a narrowly defined approval-and-queue component.

**The invariant:** The instruction selected for release must be the instruction independently approved. Changing the amount or destination cannot silently preserve the old approval.

### The component and its trust boundary

The following complete module uses Python's standard library and SQLite. It supports one organization, USD amounts stored as integer cents, a request creator, an independent reviewer, and an automated release worker. It implements creation, approval, cancellation, and an append-only execution queue.

Save it as `approval_gate.py`. Its `actor` argument must come from trusted server-side authentication, not a browser-supplied email address or arbitrary request header. The actor table is provisioned by trusted administration. Do not expose database access or role assignment to ordinary users.

This is a storage component, not a login service or payment-provider integration. A queued row means that the component authorized a fixed instruction for downstream processing; it does not mean money moved. Python documents the [SQLite connection and explicit transaction facilities](#) used here.

#### Python · `approval_gate.py` — immutable requests and transactional release

```
"""Single-organization approval storage. No payment-provider calls."""

from contextlib import contextmanager
from pathlib import Path
from typing import Iterator
import sqlite3
import time
import uuid

class ApprovalGate:
    def __init__(self, path: str | Path) -> None:
        self.db = sqlite3.connect(path, isolation_level=None, timeout=5)
        self.db.row_factory = sqlite3.Row
        self.db.execute("PRAGMA foreign_keys = ON")
        self.db.execute("PRAGMA recursive_triggers = ON")
        self.db.executescript("""
            CREATE TABLE IF NOT EXISTS actors (
                subject TEXT PRIMARY KEY,
                role TEXT NOT NULL CHECK(role IN ('maker','checker','worker')),
                active INTEGER NOT NULL CHECK(active IN (0,1))
            );
            CREATE TABLE IF NOT EXISTS requests (
```

```

        id TEXT PRIMARY KEY,
        creator TEXT NOT NULL REFERENCES actors(subject),
        amount_minor INTEGER NOT NULL CHECK(amount_minor > 0),
        currency TEXT NOT NULL CHECK(currency = 'USD'),
        destination TEXT NOT NULL,
        expires_at INTEGER NOT NULL
    );
    CREATE TABLE IF NOT EXISTS approvals (
        request_id TEXT PRIMARY KEY REFERENCES requests(id),
        approver TEXT NOT NULL REFERENCES actors(subject),
        approved_at INTEGER NOT NULL
    );
    CREATE TABLE IF NOT EXISTS cancellations (
        request_id TEXT PRIMARY KEY REFERENCES requests(id),
        cancelled_by TEXT NOT NULL REFERENCES actors(subject),
        cancelled_at INTEGER NOT NULL
    );
    CREATE TABLE IF NOT EXISTS outbox (
        request_id TEXT PRIMARY KEY REFERENCES requests(id),
        amount_minor INTEGER NOT NULL,
        currency TEXT NOT NULL,
        destination TEXT NOT NULL,
        approved_by TEXT NOT NULL REFERENCES actors(subject),
        queued_by TEXT NOT NULL REFERENCES actors(subject),
        queued_at INTEGER NOT NULL
    );
    """
    for table in ("requests", "approvals", "cancellations", "outbox"):
        for operation in ("UPDATE", "DELETE"):
            self.db.execute(f"""
                CREATE TRIGGER IF NOT EXISTS {table}_no_{operation}
                BEFORE {operation} ON {table}
                BEGIN
                    SELECT RAISE(ABORT, 'immutable record');
                END
            """)

    @contextmanager
    def _write(self) -> Iterator[None]:
        self.db.execute("BEGIN IMMEDIATE")
        try:
            yield
            self.db.execute("COMMIT")
        except BaseException:
            if self.db.in_transaction:
                self.db.execute("ROLLBACK")
            raise

    def _role(self, subject: str, allowed: tuple[str, ...]) -> str:
        actor = self.db.execute(
            "SELECT role, active FROM actors WHERE subject = ?", (subject,)
        ).fetchone()
        if actor is None or not actor["active"] or actor["role"] not in allowed:
            raise PermissionError("actor is not authorized")
        return str(actor["role"])

    def _pending(self, request_id: str) -> sqlite3.Row:
        request = self.db.execute(
            "SELECT * FROM requests WHERE id = ?", (request_id,)
        ).fetchone()
        if request is None:
            raise ValueError("request does not exist")
        if request["expires_at"] <= int(time.time()):
            raise ValueError("request has expired")
        for table in ("cancellations", "outbox"):
            exists = self.db.execute(
                f"SELECT 1 FROM {table} WHERE request_id = ?", (request_id,)
            ).fetchone()
            if exists:
                raise ValueError("request is cancelled or already queued")
        return request

    def create(
        self, actor: str, amount_minor: int, destination: str,

```

```

        lifetime_seconds: int = 3600,
    ) -> str:
        if type(amount_minor) is not int or not 1 <= amount_minor <= 10**12:
            raise ValueError("amount must be positive integer USD cents")
        if not isinstance(destination, str) or not 1 <= len(destination) <= 200:
            raise ValueError("destination must be a nonempty immutable reference")
        if type(lifetime_seconds) is not int or not 1 <= lifetime_seconds <= 86400:
            raise ValueError("lifetime must be between 1 and 86400 seconds")
        with self._write():
            self._role(actor, ("maker",))
            request_id = str(uuid.uuid4())
            self.db.execute(
                "INSERT INTO requests VALUES (?, ?, ?, 'USD', ?, ?)",
                (request_id, actor, amount_minor, destination,
                 int(time.time()) + lifetime_seconds),
            )
        return request_id

    def approve(self, actor: str, request_id: str) -> None:
        with self._write():
            self._role(actor, ("checker",))
            request = self._pending(request_id)
            if request["creator"] == actor:
                raise PermissionError("self-approval is forbidden")
            self.db.execute(
                "INSERT INTO approvals VALUES (?, ?, ?)",
                (request_id, actor, int(time.time())),
            )

    def cancel(self, actor: str, request_id: str) -> None:
        with self._write():
            role = self._role(actor, ("maker", "checker"))
            request = self._pending(request_id)
            if role == "maker" and request["creator"] != actor:
                raise PermissionError("cannot cancel another maker's request")
            self.db.execute(
                "INSERT INTO cancellations VALUES (?, ?, ?)",
                (request_id, actor, int(time.time())),
            )

    def queue(self, actor: str, request_id: str) -> dict[str, object]:
        with self._write():
            self._role(actor, ("worker",))
            request = self._pending(request_id)
            approval = self.db.execute(
                "SELECT * FROM approvals WHERE request_id = ?", (request_id,)
            ).fetchone()
            if approval is None:
                raise PermissionError("request has no approval")
            self._role(approval["approver"], ("checker",))
            if approval["approver"] == request["creator"]:
                raise PermissionError("self-approval is forbidden")
            self.db.execute(
                "INSERT INTO outbox VALUES (?, ?, ?, ?, ?, ?, ?)",
                (request_id, request["amount_minor"], request["currency"],
                 request["destination"], approval["approver"], actor,
                 int(time.time())),
            )
            job = dict(self.db.execute(
                "SELECT * FROM outbox WHERE request_id = ?", (request_id,)
            ).fetchone())
        return job

    def close(self) -> None:
        self.db.close()

```

## Why the gate changes the outcome

**The reviewed record cannot be edited in place.** Database triggers reject updates and deletions on submitted requests. The approval points to a fixed amount, currency, destination reference, creator, and

deadline. There is no editable approved flag attached to changing instructions. Recursive triggers are enabled because [SQLite documents that replacement operations fire delete triggers only with that setting](#).

**Different role names are not the only check.** The code also compares the creator and approver identities. That prevents the creator from approving the same request after their role changes. In a deployed system, identities should resolve to stable personnel identifiers; two accounts controlled by the same person do not provide genuine independence.

**The release decision uses current local authorization state.** The reviewer must still have an active checker role when the worker queues the request. Cancelled, expired, unapproved, or previously queued requests are rejected.

**Validation and queue insertion share a write transaction.** SQLite documents that [BEGIN IMMEDIATE starts a write transaction immediately](#) and that only one write transaction can run at a time. Here, competing changes through the same database cannot interleave between those checks and insertion. The outbox's primary key additionally permits only one queue record per request identifier. A busy or failed transaction produces an error rather than an authorized job.

**The queued payload is copied from the approved record.** The worker does not receive an amount or destination from an untrusted release request. It retrieves them from the immutable submission.

**Immutability has a boundary.** These triggers protect records against normal updates; they do not protect against an administrator who can replace the database file or remove the triggers. SQLite does not provide separate application-user database accounts. Protect the file and service, restrict administrative access, and keep independently protected audit evidence.

## Verify the security properties

A successful approval screen is not an adequate acceptance test. The important questions concern actions that must fail. [OWASP's workflow-testing guidance](#) recommends misuse cases that exercise skipped, repeated, or reordered business steps.

Save this as `check_approval_gate.py` beside the module. It creates a real temporary database, provisions fictitious actors, exercises the component, and removes the temporary database afterward. The role changes are trusted test setup, not a user-facing feature.

### Python · `check_approval_gate.py` — verify rejected and permitted operations

```
from pathlib import Path
from tempfile import TemporaryDirectory
import sqlite3

from approval_gate import ApprovalGate

def blocked(label, operation, expected):
    try:
        operation()
    except expected:
        print(f"PASS: {label}")
    else:
        raise AssertionError(f"Unexpectedly allowed: {label}")
```

```
with TemporaryDirectory() as directory:
    gate = ApprovalGate(Path(directory) / "approvals.sqlite3")
    try:
        gate.db.executemany(
            "INSERT INTO actors VALUES (?, ?, 1)",
            [("maker@example.com", "maker"),
             ("checker@example.com", "checker"),
             ("worker@example.com", "worker")],
        )
        request = gate.create("maker@example.com", 25000, "destination-original-v1")
        blocked("unapproved release", lambda: gate.queue(
            "worker@example.com", request), PermissionError)

        gate.db.execute("UPDATE actors SET role = 'checker' WHERE subject = ?",
            ("maker@example.com",))
        blocked("self-approval after a role change", lambda: gate.approve(
            "maker@example.com", request), PermissionError)
        gate.db.execute("UPDATE actors SET role = 'maker' WHERE subject = ?",
            ("maker@example.com",))

        gate.approve("checker@example.com", request)
        blocked("post-approval edit", lambda: gate.db.execute(
            "UPDATE requests SET amount_minor = 2500000 WHERE id = ?",
            (request,)), sqlite3.IntegrityError)

        job = gate.queue("worker@example.com", request)
        assert job["amount_minor"] == 25000
        assert job["destination"] == "destination-original-v1"
        print("PASS: queued payload matches the approved request")
        blocked("duplicate release", lambda: gate.queue(
            "worker@example.com", request), ValueError)

        cancelled = gate.create("maker@example.com", 25000, "destination-original-v1")
        gate.approve("checker@example.com", cancelled)
        gate.cancel("maker@example.com", cancelled)
        blocked("cancelled release", lambda: gate.queue(
            "worker@example.com", cancelled), ValueError)
    finally:
        gate.close()
```

Run the checks:

#### Bash · run the approval-integrity checks

```
python3 check_approval_gate.py
```

The expected output is:

#### Plaintext · expected approval-gate results

```
PASS: unapproved release
PASS: self-approval after a role change
PASS: post-approval edit
PASS: queued payload matches the approved request
PASS: duplicate release
PASS: cancelled release
```

Before integrating the pattern into an application, also test expired requests, revoked reviewers, two simultaneous workers, cancellation racing with release, cross-organization access, and every alternate write path. The temporary checks above do not test your authentication service, user interface, or payment provider.

The reviewer interface also matters. It must display the significant fields from the exact immutable request being approved, not a cached summary from a different version. An identifier that cannot be edited is useful only when the human sees what that identifier actually represents.

## Protect the execution boundary

### An outbox is not proof of payment

An outbox is a durable record of work to perform. It helps avoid losing the instruction between a database decision and a later external action. [AWS describes the transactional outbox pattern](#) and the need to handle duplicate delivery.

The unique request identifier in this component prevents a second local queue entry for that request. It does not guarantee one external transfer. A worker can send a request successfully, lose the response, and retry. A complete integration must persist provider operation identifiers, reconcile uncertain outcomes, and use the provider's supported duplicate-prevention mechanism.

For example, [Stripe documents idempotency keys, parameter matching, and retention behavior](#). Apply the rules for the actual provider, account, endpoint, and API version. Do not assume that every provider remembers an idempotency key forever, or generate a fresh key merely because the outcome of an earlier attempt is uncertain.

### A fixed reference must have a fixed meaning

The example's destination strings are fictitious references, not bank account numbers or usable payment tokens. In an integration, the reference must resolve to the exact approved destination. If a provider's beneficiary object is editable, retain an immutable internal destination version and verify the resolved instruction before dispatch.

Do not rebuild a payment from a vendor's current details after approval. Similarly, an infrastructure worker should not rebuild an approved change from an editable ticket description. Its targets, environment, privileges, and change content need an equally precise binding.

### Expiry and cancellation need a defined boundary

This component checks expiry and reviewer authority when it queues a request. It refuses cancellation after queueing. It does not claim that disabling a reviewer later will recall an already queued or externally accepted instruction.

A production dispatcher therefore needs an explicit policy for maximum queue age, dispatch deadlines, emergency holds, cancellation acknowledgments, and uncertain provider results. Decide which events require fresh authorization and where cancellation stops being technically enforceable. Do not present a "cancelled" interface state until the relevant system has actually confirmed the cancellation.

### The business policy still needs an owner

The module's amount bound is an input-validation limit, not a spending-authority policy. Real approval limits, allowed destinations, business hours, exceptional transactions, and escalation rules must come from the organization. One valid request identifier also does not prevent two different requests for the same invoice; that requires a separate business-level uniqueness rule.

For a shared application, bind organization identity throughout requests, approvals, roles, and execution records. Add object-level permissions to the surrounding API. Apply [cross-site request forgery protection](#) to approval actions where cookie-based authentication is used. Do not mistake this single-organization component for a complete multi-tenant authorization system.

## Potential business and infrastructure repercussions

The following are possible consequences of the worked scenario and closely related approval failures, not claims about an incident at Sunimod or estimates of average losses:

- **Unauthorized payments or inflated refunds.** In the example, a \$250 approval is attached to a \$25,000 instruction. That is \$24,750 above the reviewed amount, before considering the separate destination change.
- **Cash-flow and operational disruption.** A business may need to pause payment processing, reconcile affected transactions, and handle legitimate obligations while investigating the discrepancy.
- **Unreviewed infrastructure changes.** Applied to change management, the same design flaw could attach approval for a limited change to a broader privilege grant, a different production target, or a more disruptive configuration.
- **Data disclosure.** In an export-approval workflow, changing the recipient or the selected dataset after review could cause the system to release information outside the approved scope.
- **Misleading audit history.** A reviewer's name beside a mutable request can incorrectly suggest approval of the final details. Investigators may struggle to establish what was displayed, what was authorized, and what was executed.
- **Recovery work and damaged confidence.** Staff may need to reconstruct records, correct systems, explain discrepancies to customers or vendors, and reassess whether other approvals can be trusted.

Assess exposure using your actual workflow: who can change consequential fields, which downstream actions are reachable, what limits exist, how long approvals remain usable, and whether execution can be reversed. A small internal workflow and a high-value automated payment system should not receive identical risk treatment.

## Turn the fix into a bounded business project

A useful project starts with one consequential workflow, not a promise to redesign every system. Choose vendor payments, refunds, account privileges, data exports, or production changes. Assign a business owner who can define what must be independently reviewed and a technical owner who can enforce that rule.

### Map the real approval path

Trace creation, review, editing, imports, integrations, queueing, dispatch, cancellation, and reconciliation. Identify every field that changes the business meaning of the action. Include related records: an unchanged request can still resolve to a changed destination or target.

## Replace status checks with enforced rules

Implement immutable submissions or rigorously controlled versions, independent approver checks, defined expiry, and a release path that consumes the reviewed instruction. Make exceptions explicit, time-limited, and attributable to an authorized risk owner. Urgency should not silently turn a two-person process into a one-person process.

Small teams may have limited staffing, but one person using two accounts is not an equivalent safeguard. The business owner must choose a genuine independent reviewer, an appropriate external control, or a documented acceptance of the remaining risk.

## Require evidence that the control works

- Every consequential post-review change is blocked or requires a new approval.
- The creator cannot approve the same instruction, including after a role change.
- Cancelled, expired, unapproved, and duplicate release attempts fail as designed.
- The dispatched payload matches the reviewed payload, including referenced destinations and targets.
- Concurrent execution and ambiguous provider responses have tested outcomes.
- Review, release, exception, and provider records can be reconciled by request identifier.

Measure these as acceptance criteria, not as assurances implied by a green status label. Review them again when an integration, import path, business policy, or administrative permission changes.

## Detect and investigate approval mismatches

For this workflow, useful events include request creation, the approved request identifier, reviewer identity, cancellation, release, policy rejection, and the downstream operation identifier. Alert on attempted edits to approved instructions, release without matching approval, repeated self-approval attempts, and discrepancies between queued and provider-confirmed details.

[OWASP's logging guidance](#) recommends recording application-level security events and protecting the resulting records. Keep secrets and unnecessary bank or payment-card details out of general logs; use restricted evidence records and non-secret correlation identifiers.

If a mismatch is discovered, follow the incident process for the affected system: hold the implicated execution path where safe, preserve review and execution evidence, establish what the downstream system accepted, and involve the responsible business and technical owners. Do not treat the approval screen alone as proof of what was authorized, and do not make another live transaction merely to reproduce the issue.

**The lasting lesson:** Approval is not a reusable badge attached to a record. It is a bounded authorization for a particular action. A trustworthy workflow preserves that meaning from review through execution.

## Sources and further reading

- [NIST: Separation of Duty](#) — independent duties and the two-person principle.
- [OWASP: Transaction Authorization Cheat Sheet](#) — significant transaction details, modification protection, and execution checks.
- [OWASP: Authorization Cheat Sheet](#) — server-side permissions and authorization boundaries.
- [OWASP: Business Logic Vulnerability](#) — workflow abuse, including changes after approval.
- [OWASP: Testing for the Circumvention of Work Flows](#) — business-specific misuse tests.
- [MITRE: CWE-367](#) — the distinct time-of-check/time-of-use race pattern.
- [SQLite: Transaction Control](#) and [Python: sqlite3](#) — the database and transaction mechanisms used in the code.
- [AWS: Transactional Outbox Pattern](#) — durable handoff and duplicate-delivery considerations.
- [Stripe: Idempotent Requests](#) — a provider-specific example of retry semantics.
- [OWASP: Logging Cheat Sheet](#) — security events, sensitive-data exclusions, and log protection.

## Make your approval workflow enforce the decision you intended

Hire Sunimod to review and harden the approval paths in your custom application, internal portal, or business automation. A focused project can map the workflow, identify mutable approval gaps, implement the appropriate controls, and verify that changed instructions cannot reuse old permission.

Describe the workflow, the systems involved, who creates and approves requests, and what happens after approval. Ask for an approval-workflow hardening project with clear scope and acceptance tests.

[Request an approval-workflow project quote](#)

Do not include passwords, API keys, payment-card details, or banking secrets in your request.