



SUNIMOD FIELD NOTES

The Payment You Approved Can Change Before It Runs: How Stale Approvals Bypass Business Controls

An approved payment can change before release. Learn how stale approvals undermine business controls and how version-bound review prevents unapproved changes.

BY Christopher Jacobson

PUBLISHED October 15, 2026

TOPIC CISSP, Domain 2

<https://sunimod.com/stale-payment-approvals-business-control-bypass/>

The central lesson: Approval belongs to a specific version of a business action, not to a record that can change underneath it. When a significant detail changes, the old approval must stop authorizing the new action.

- [Where the business control breaks](#)
- [The attack and a local reproduction](#)
- [Potential business repercussions](#)
- [A stronger approval design](#)
- [Working code and verification](#)
- [A practical project for your business](#)

When approval loses its meaning

Consider a business rule: one employee prepares a supplier payment, and another authorized employee reviews it before release. The intended protection is that the preparer cannot independently decide where company money goes.

That rule fails when the application remembers only that a reviewer clicked **Approve**. A timestamp and an approver name do not, by themselves, establish which amount, recipient, or destination the reviewer accepted.

This is a governance problem expressed through software: the written policy promises independent review, but the implementation does not preserve the decision through execution. [NIST describes separation of duty](#) as limiting the privileges one user can combine to misuse a system. In our payment example, allowing the preparer to change the action after review defeats that objective without granting them an approver role.

Authentication and authorization also answer different questions. A valid login identifies a user; it does not establish that every action available through the application is permitted. [OWASP's authorization guidance](#) makes that distinction and recommends checking permissions on every request.

Terminology: This article uses *stale approval* for an approval that no longer matches the current action. It is a descriptive label, not a claim about a particular product or a newly discovered vulnerability.

How stale approvals are exploited

The following scenario is fictitious. Assume the application lets a preparer edit a payment's destination after review and that its release job checks only an approval flag. Those permissions and behaviors are prerequisites; a system that already invalidates the approval would stop this sequence.

1. **Prepare an ordinary request.** An employee creates a \$4,800 payment to the verified destination for Supplier A.
2. **Obtain legitimate review.** A finance reviewer checks the amount and recipient. The application sets `approved = 1`.

3. **Change the destination.** A dishonest preparer, or someone controlling the preparer's account, switches the destination to another account reference. The application leaves the approval intact.
4. **Let the normal release job run.** The job selects approved records and reads their current values, including the substituted destination.
5. **Produce an unauthorized instruction.** The job submits the changed instruction under the authority of the earlier review. Whether money actually moves also depends on the payment provider's controls.

The reviewer did not approve the substituted destination. The software nevertheless treats the old decision as permission for a different action. This is an example of the broader workflow-enforcement problem described by [MITRE's CWE-841](#): required business steps can be bypassed or performed in an invalid sequence.

Reproduce the broken approval flag

This complete Python example creates a real, temporary in-memory SQLite database. It approves a record, changes its destination, and then runs the selection a poorly designed release job might use. The identifiers are fictitious; nothing connects to a bank or sends a payment.

Keep testing isolated. Use the local examples or an explicitly authorized test environment. Do not alter real supplier records or live payment instructions to find out whether your controls work.

Python · `approval_drift.py` — an approval flag survives a material change

```
import sqlite3

with sqlite3.connect(":memory:") as db:
    db.execute("CREATE TABLE payments (id INTEGER PRIMARY KEY, "
               "amount_minor INTEGER, destination_ref TEXT, approved INTEGER)")
    db.execute("INSERT INTO payments VALUES (?, ?, ?, ?)",
               (41, 480000, "vendor_a_v1", 0))
    db.execute("UPDATE payments SET approved = 1 WHERE id = ?", (41,))
    db.execute("UPDATE payments SET destination_ref = ? WHERE id = ?",
               ("vendor_b_v1", 41))
    row = db.execute("SELECT amount_minor, destination_ref "
                     "FROM payments WHERE id = ? AND approved = 1", (41,)).fetchone()
    print(f"Still eligible: {row is not None}")
    print(f"Amount in cents: {row[0]}")
    print(f"Destination selected: {row[1]}")
```

Plaintext · result of the local reproduction

```
Still eligible: True
Amount in cents: 480000
Destination selected: vendor_b_v1
```

The important result is `Still eligible: True`. The destination was changed after approval, but the release query still accepts the record.

Notice that the example uses parameterized SQL rather than concatenating user input into a query. Parameter binding is valuable, but it does not supply the missing business rule. The defect is that changing a payment does not invalidate its approval. Python's [SQLite documentation](#) explains parameter binding and the database interface used here.

A stale approval is not always a race condition

The example does not require precise timing. A record could be approved in the morning, edited after lunch, and released that evening.

A related timing defect occurs when a worker checks approval, another process changes the record, and the worker then reads the changed details. That check/use gap fits [MITRE's description of a time-of-check/time-of-use race condition](#). Moving the two reads closer together is not the same as making the operation atomic.

Both cases need a stable relationship between the approved data and the action. The concurrent case additionally requires a transaction or equivalent coordination so competing changes cannot slip between the final check and the creation of the authorized instruction.

The linked-record trap

In this design, freezing the payment row would still be insufficient if it held only a mutable supplier identifier. A worker that resolves "Supplier A's current bank account" at release time could obtain different banking details from those the reviewer saw.

The proposed fix is to bind the request to an immutable, verified destination version. A banking-detail change creates a new version rather than silently changing the meaning of an existing reference. The review screen must display the significant details behind that version, not merely an opaque database ID.

Potential repercussions for your business and infrastructure

The consequences below are scenario-based possibilities, not statistics or claims that every vulnerable system will suffer every outcome. The impact depends on what the workflow can change and which independent controls remain in place.

How a broken approval relationship can become a business problem

Exposure	Potential consequence
Misdirected payments	Funds could go to an unintended recipient while the legitimate supplier remains unpaid. The business may need to resolve the missing payment while maintaining enough cash for normal operations.
Operational disruption	Investigating suspicious instructions could require holding payment batches, rechecking supplier details, and manually reviewing previously approved work.
Unauthorized infrastructure changes	The same design flaw in a change-management tool could allow a reviewed configuration to be replaced with one that exposes a service, broadens access, or changes a production setting.
Unreliable decision history	An audit screen might associate a reviewer with data they never saw. Without preserved versions, reconstructing who approved what becomes harder.
Commercial and relationship costs	Customer or supplier disputes, incident investigation, corrective engineering, and review of applicable contractual obligations could follow. Specific legal or notification duties depend on the actual incident and jurisdiction.

For a useful internal risk estimate, examine the value and number of requests awaiting release, who can still edit them, and how long a mismatch could remain unnoticed. Also identify the workflow owner: someone must be accountable for deciding the approval rules and accepting any exceptions.

The business objective is not simply to add friction. It is to preserve the reviewer's decision while allowing legitimate corrections to move through a clear, repeatable reapproval process.

Design an approval that cannot drift

Start with an enforceable rule: release only the exact reviewed revision, within its validity period, under the authority of an eligible independent reviewer. A material edit produces a new revision that needs a new decision.

[OWASP's transaction-authorization guidance](#) calls for meaningful review of significant transaction data, server-side enforcement, protection against changes, and a final authorization check tied to execution. Those principles provide the foundation; your business must define the fields, limits, identities, and exceptions.

- **Bind the decision to the action.** Preserve the request ID, revision, amount, currency, destination version, reviewer, and validity period. Include every additional field that can change the business effect.
- **Separate preparation from review.** Compare canonical person identities, not just usernames. Two accounts controlled by one person do not provide independent review.
- **Make edits invalidate permission.** Increase the revision and clear the active approval together. A browser submitting an older revision must receive a conflict, not silently approve newer data.
- **Control the release transition.** Check the approved snapshot and create the release instruction in one database transaction. Enforce one local instruction per request.
- **Preserve evidence.** Retain reviewed versions and record the relevant actor, decision, revision, and time without exposing credentials or unnecessary banking data.

Apply the rule to every path: the normal screen, administrative tools, imports, integrations, and scheduled jobs. Hiding an edit button does not protect an independently callable endpoint. A documented exception also needs its own constrained workflow; an unrestricted "force approved" switch recreates the original problem.

A state-oriented design makes these rules testable. [OWASP's business-logic guidance](#) recommends explicit server-side workflow states, protection against concurrent operations, and testing that reflects the intended business process.

Implement the server-side approval gate

The complete module below implements persistent requests, revision changes, independent approval, expiration checks, event recording, and one-time admission to an outbox. An outbox is a database table holding authorized instructions for a separate execution service.

This is an internal approval-storage component, not a login system or payment processor. It does not make bank requests or report fictitious payment success. Its scope is one organization, USD payments,

one independent reviewer, and revisions made only by the original preparer. The ten-minute approval lifetime and \$1,000,000 input ceiling are explicit example choices, not recommended limits for your business; an input ceiling is not a reviewer-specific spending policy.

Identity and storage are trust boundaries. The `actor` argument must come from authenticated server-side identity, never an identity supplied in a form or unverified header. Provision the `people` table through trusted administration using canonical person IDs. Keep the database outside the web root, in a service-owned directory with restrictive permissions. Anyone who can directly rewrite its tables can bypass these application controls.

The `destination_ref` must resolve to a verified, immutable destination version in your payment or vault system. The module validates the reference's format; it does not establish destination ownership or make a mutable external record immutable. Those checks belong in the trusted integration and review interface.

Save this as `approval_store.py`. It uses Python's standard library and is suitable for running the example on Ubuntu with Python 3.12.

Python · `approval_store.py` — revision-bound approval and atomic queue admission

```
import json
import re
import sqlite3
import time
import uuid
from contextlib import closing, contextmanager
from pathlib import Path
from typing import Iterator

SCHEMA = """
CREATE TABLE IF NOT EXISTS people (
    id TEXT PRIMARY KEY,
    role TEXT NOT NULL CHECK (role IN ('prepare', 'approve', 'release')),
    active INTEGER NOT NULL CHECK (active IN (0, 1))
);
CREATE TABLE IF NOT EXISTS requests (
    id TEXT PRIMARY KEY,
    creator_id TEXT NOT NULL REFERENCES people(id),
    revision INTEGER NOT NULL CHECK (revision > 0),
    payload TEXT NOT NULL,
    approver_id TEXT REFERENCES people(id),
    approved_revision INTEGER,
    approved_payload TEXT,
    expires_at INTEGER,
    queued INTEGER NOT NULL DEFAULT 0 CHECK (queued IN (0, 1))
);
CREATE TABLE IF NOT EXISTS outbox (
    request_id TEXT PRIMARY KEY REFERENCES requests(id),
    revision INTEGER NOT NULL,
    payload TEXT NOT NULL,
    approver_id TEXT NOT NULL REFERENCES people(id),
    release_actor_id TEXT NOT NULL REFERENCES people(id),
    authorized_at INTEGER NOT NULL,
    expires_at INTEGER NOT NULL
);
CREATE TABLE IF NOT EXISTS events (
    id INTEGER PRIMARY KEY,
    request_id TEXT NOT NULL REFERENCES requests(id),
    actor_id TEXT NOT NULL REFERENCES people(id),
    action TEXT NOT NULL,
    revision INTEGER NOT NULL,
    payload TEXT NOT NULL,
    occurred_at INTEGER NOT NULL
);

```

```

"""

class ApprovalStore:
    """Internal, single-organization USD approval storage; no payment transport."""

    def __init__(self, path: str | Path):
        self.path = str(path)
        with closing(self._connect()) as db:
            db.executescript(SCHEMA)

    def _connect(self) -> sqlite3.Connection:
        db = sqlite3.connect(self.path, timeout=5, isolation_level=None)
        db.row_factory = sqlite3.Row
        db.execute("PRAGMA foreign_keys = ON")
        return db

    @contextmanager
    def _transaction(self) -> Iterator[sqlite3.Connection]:
        with closing(self._connect()) as db:
            try:
                db.execute("BEGIN IMMEDIATE")
                yield db
                db.execute("COMMIT")
            except BaseException:
                if db.in_transaction:
                    db.execute("ROLLBACK")
                raise

    @staticmethod
    def _require(db: sqlite3.Connection, actor: str, role: str) -> None:
        person = db.execute(
            "SELECT role, active FROM people WHERE id = ?", (actor,)
        ).fetchone()
        if person is None or not person["active"] or person["role"] != role:
            raise PermissionError("Active role required: " + role)

    @staticmethod
    def _load(db: sqlite3.Connection, request_id: str) -> sqlite3.Row:
        row = db.execute(
            "SELECT * FROM requests WHERE id = ?", (request_id,)
        ).fetchone()
        if row is None:
            raise LookupError("Request not found")
        return row

    @staticmethod
    def _check_revision(row: sqlite3.Row, expected: int) -> None:
        if row["queued"]:
            raise ValueError("Request already queued")
        if type(expected) is not int or row["revision"] != expected:
            raise ValueError("Revision changed; review the current request")

    @staticmethod
    def _payload(amount_minor: int, destination_ref: str) -> str:
        if type(amount_minor) is not int or not 1 <= amount_minor <= 100_000_000:
            raise ValueError("USD amount must be 1 to 100000000 cents")
        if not isinstance(destination_ref, str) or not re.fullmatch(
            r"[A-Za-z0-9_-]{1,100}", destination_ref
        ):
            raise ValueError("Invalid immutable destination reference")
        return json.dumps(
            {"amount_minor": amount_minor, "currency": "USD",
             "destination_ref": destination_ref},
            sort_keys=True, separators=(",", ":")
        )

    @staticmethod
    def _event(db: sqlite3.Connection, row: sqlite3.Row,
              actor: str, action: str) -> None:
        db.execute(
            "INSERT INTO events "
            "(request_id, actor_id, action, revision, payload, occurred_at) "
            "VALUES (?, ?, ?, ?, ?, ?)",

```

```

        (row["id"], actor, action, row["revision"],
         row["payload"], int(time.time()))
    )

def create(self, actor: str, amount_minor: int, destination_ref: str) -> str:
    payload = self._payload(amount_minor, destination_ref)
    request_id = uuid.uuid4().hex
    with self._transaction() as db:
        self._require(db, actor, "prepare")
        db.execute(
            "INSERT INTO requests (id, creator_id, revision, payload) "
            "VALUES (?, ?, 1, ?)", (request_id, actor, payload)
        )
        self._event(db, self._load(db, request_id), actor, "created")
    return request_id

def revise(self, actor: str, request_id: str, expected: int,
           amount_minor: int, destination_ref: str) -> int:
    payload = self._payload(amount_minor, destination_ref)
    with self._transaction() as db:
        self._require(db, actor, "prepare")
        row = self._load(db, request_id)
        self._check_revision(row, expected)
        if row["creator_id"] != actor:
            raise PermissionError("Only the original preparer may revise")
        db.execute(
            "UPDATE requests SET revision = revision + 1, payload = ?, "
            "approver_id = NULL, approved_revision = NULL, "
            "approved_payload = NULL, expires_at = NULL WHERE id = ?",
            (payload, request_id)
        )
        updated = self._load(db, request_id)
        self._event(db, updated, actor, "revised")
    return updated["revision"]

def approve(self, actor: str, request_id: str, expected: int) -> None:
    with self._transaction() as db:
        self._require(db, actor, "approve")
        row = self._load(db, request_id)
        self._check_revision(row, expected)
        if row["creator_id"] == actor:
            raise PermissionError("Self-approval is forbidden")
        db.execute(
            "UPDATE requests SET approver_id = ?, "
            "approved_revision = revision, approved_payload = payload, "
            "expires_at = ? WHERE id = ?",
            (actor, int(time.time()) + 600, request_id)
        )
        self._event(db, row, actor, "approved")

def queue(self, actor: str, request_id: str, expected: int) -> str:
    with self._transaction() as db:
        self._require(db, actor, "release")
        row = self._load(db, request_id)
        self._check_revision(row, expected)
        if (row["approved_revision"] != row["revision"]
            or row["approved_payload"] != row["payload"]):
            raise PermissionError("Current revision needs approval")
        self._require(db, row["approver_id"], "approve")
        if row["approver_id"] == row["creator_id"]:
            raise PermissionError("Self-approval is forbidden")
        now = int(time.time())
        if row["expires_at"] is None or row["expires_at"] <= now:
            raise PermissionError("Approval expired")
        db.execute(
            "INSERT INTO outbox VALUES (?, ?, ?, ?, ?, ?, ?)",
            (request_id, row["revision"], row["approved_payload"],
             row["approver_id"], actor, now, row["expires_at"])
        )
        db.execute("UPDATE requests SET queued = 1 WHERE id = ?",
                   (request_id,))
        self._event(db, row, actor, "queued")
    return request_id

```

Why this closes the demonstrated gap

`revise()` increments the revision and removes the active approval in the same transaction. The event history retains the payloads associated with successful changes and reviews. `approve()` accepts only the revision the reviewer actually submitted.

`queue()` compares both the approved revision and the approved payload with the current record. It also checks that the reviewer remains active in the approval role and that the approval has not expired. The instruction contains the stored approved payload rather than details supplied by the release caller.

The final check, outbox insert, state change, and success event occur inside `BEGIN IMMEDIATE`. [SQLite documents](#) that this begins a write transaction immediately and may encounter contention from another writer. In this implementation, a lock timeout or database error is propagated; it does not become permission to continue.

The outbox's primary key prevents a second local instruction for the same request. Through this module, an edit after queue admission is rejected rather than changing an instruction already handed off. These properties are limited to this database boundary; they do not prove exactly-once execution at an external provider.

Verify stale reviews and concurrent release

Save the following as `verify_approval_store.py` beside the module. It uses fictitious identities in a temporary database and performs actual database operations. The direct identity inserts are test-fixture administration, not a public account-provisioning interface.

Python · `verify_approval_store.py` — reject stale decisions and racing queue attempts

```
import json
import sqlite3
import tempfile
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
from threading import Barrier

from approval_store import ApprovalStore

with tempfile.TemporaryDirectory() as directory:
    path = Path(directory) / "approvals.sqlite3"
    store = ApprovalStore(path)
    with sqlite3.connect(path) as db:
        db.executemany("INSERT INTO people VALUES (?, ?, 1)", [
            ("person-001", "prepare"),
            ("person-002", "approve"),
            ("service-001", "release"),
        ])

    request_id = store.create("person-001", 480000, "vendor_a_v1")
    store.approve("person-002", request_id, 1)
    store.revise("person-001", request_id, 1, 480000, "vendor_b_v1")

    try:
        store.queue("service-001", request_id, 2)
    except PermissionError:
        print("Blocked: changed payment has no current approval")
    else:
        raise AssertionError("Stale approval was accepted")

    try:
        store.approve("person-002", request_id, 1)
```

```
except ValueError:
    print("Blocked: reviewer submitted an old revision")
else:
    raise AssertionError("Old review was accepted")

store.approve("person-002", request_id, 2)

start = Barrier(2)

def enqueue() -> str:
    start.wait(timeout=5)
    try:
        store.queue("service-001", request_id, 2)
        return "queued"
    except ValueError:
        return "rejected"

with ThreadPoolExecutor(max_workers=2) as pool:
    outcomes = list(pool.map(lambda _: enqueue(), range(2)))
assert sorted(outcomes) == ["queued", "rejected", outcomes

with sqlite3.connect(path) as db:
    rows = db.execute("SELECT revision, payload FROM outbox").fetchall()
assert len(rows) == 1 and rows[0][0] == 2
assert json.loads(rows[0][1])["destination_ref"] == "vendor_b_v1"
print("Confirmed: concurrent release attempts created one approved snapshot")
```

Bash · run the verification on Ubuntu

```
python3 verify_approval_store.py
```

Plaintext · successful verification output

```
Blocked: changed payment has no current approval
Blocked: reviewer submitted an old revision
Confirmed: concurrent release attempts created one approved snapshot
```

The meaningful difference: Changing the destination no longer carries the old permission forward. The new revision becomes eligible only after a fresh review, and competing queue attempts produce one local instruction.

For your application, expand the acceptance tests to cover self-approval after a role change, disabled reviewers, expired decisions, unauthorized editors, every material field, imports, administrative routes, and failures while recording a release. Also test that a second request for the same invoice cannot bypass any invoice-level duplicate-payment rule.

Passing these local checks is evidence for specific behaviors, not a certification of the surrounding system. Generic scanning also cannot establish all of an application's intended business rules; [OWASP recommends combining tools with business-process review and testing](#).

Carry the control through actual execution

The application is not finished when an approved row reaches an outbox. The execution service must preserve the same decision when it calls the real payment provider or infrastructure API.

Do not rebuild the instruction from mutable data

Use the queued snapshot and its immutable destination version. Do not replace it with the supplier's current default account or merge it with fresh browser parameters. Before dispatch, enforce the applicable approval deadline and cancellation or revocation policy; the module's check occurs at queue admission, not at an arbitrarily later network call.

If the external system cannot address a stable destination version, the integration needs another way to preserve and verify the reviewed destination. Treat that as a design requirement rather than assuming the reference name provides a guarantee.

Handle retries without inventing certainty

A local database commit and a remote provider call are different operations. A timeout can leave the caller uncertain whether the provider accepted an instruction. Use the provider's documented idempotency and status-reconciliation mechanisms where available, with a stable identifier for the same instruction. Do not generate a new payment identity on every retry.

This follows [OWASP's guidance on idempotency for external actions](#). A unique outbox row does not prevent a poorly implemented worker from submitting that row twice, and it does not prevent a separate request from representing the same invoice.

Keep a decision history you can investigate

The example's `events` table records successful transitions. It is not tamper-proof, and rejected attempts raise exceptions rather than writing durable denial events. The surrounding service should record those denials and administrative role changes in a protected logging destination outside the business transaction being rolled back.

[OWASP's logging guidance](#) recommends recording authorization failures and relevant event context, protecting log access, and avoiding inappropriate sensitive data. For this workflow, a useful investigation should connect the original request, reviewed revision, later edits, release decision, and provider outcome.

Residual risk still matters. These controls do not prevent two reviewers from colluding, a reviewer from knowingly approving fraud, or a sufficiently privileged administrator from rewriting the application and its records. Independent verification, tightly controlled administrative access, reconciliation, and protected evidence address different parts of that risk.

Turn the gap into a scoped business project

A practical starting point is one consequential workflow: supplier payments, refunds, privileged-access changes, or production configuration approvals. The project is an **approval-integrity retrofit**, not an automatic commitment to replace the entire business system.

First, map who can prepare, modify, review, and release each action. Then trace what the reviewer sees, what the database preserves, and what the downstream job actually executes. The most important finding may be a single alternate route that bypasses otherwise sound checks.

The implementation scope should produce a written approval rule, revision-bound decisions, a controlled release path, integration-specific retry handling, and evidence from tests of the ways the rule can fail. Assign an owner for exceptions and ongoing changes so the protection does not disappear during the next workflow update.

Sunimod's [custom software and integration services](#) include workflow automation, role-aware access, APIs, and modernization. Hire Sunimod to assess the approval path in your existing application and scope the changes needed to make its behavior match your business rules.

Sources and further reading

- [NIST: Separation of Duty](#) — the principle behind independent preparation and review.
- [OWASP: Transaction Authorization Cheat Sheet](#) — significant transaction data, modification controls, and execution authorization.
- [OWASP: Authorization Cheat Sheet](#) — permissions, enforcement, and authorization testing.
- [MITRE: CWE-841, Improper Enforcement of Behavioral Workflow](#), and [CWE-367, Time-of-check Time-of-use Race Condition](#) — related weakness descriptions.
- [OWASP: Business Logic Security Cheat Sheet](#) — workflow states, concurrency, external actions, and process-aware testing.
- [SQLite: Transactions](#) — write transactions, contention, commits, and rollback behavior.
- [Python 3.12: sqlite3](#) — the standard-library database interface and parameter binding.
- [OWASP: Logging Cheat Sheet](#) — event context, authorization failures, and protection of logs.

Make “approved” mean the action your business actually reviewed

A green checkmark should not survive a changed payment destination or authorize an unreviewed production change. Bring Sunimod the workflow, the systems it connects to, and the rules your team needs enforced. We can scope a focused project to review the gaps, implement stronger controls, and verify the critical paths.

[Request an approval-workflow quote from Sunimod](#)

Describe the business process and the outcome you need. Do not submit passwords, API keys, banking details, or customer records through the quote form.