



SUNIMOD FIELD NOTES

Approval Bypass: When Two Accounts Are Still One Person

A workflow can log two usernames and still let one person authorize a risky change. Learn how to detect self-approval and build enforceable, auditable two-person controls.

BY Christopher Jacobson

PUBLISHED October 15, 2026

TOPIC CISSP, Domain 2

<https://sunimod.com/approval-bypass-broken-separation-of-duties/>

The central lesson: Two accounts are not necessarily two independent people. A meaningful approval must connect an authorized reviewer to the exact action being approved, keep that reviewer independent of the requester, and remain valid when the action takes effect.

In this guide

1. [What separation of duties protects](#)
2. [How the approval bypass works](#)
3. [See the faulty check in code](#)
4. [Build an enforceable approval boundary](#)
5. [Test the control, not just the button](#)
6. [Potential business repercussions](#)
7. [A practical Sunimod project](#)

What separation of duties protects

Separation of duties divides sensitive responsibilities so that one person cannot complete an entire risky process alone. NIST distinguishes between preventing conflicting role assignments in advance and enforcing separation when a particular operation occurs. The second approach can let employees perform different duties on different requests without allowing them to approve their own work. [1]

Consider a purchasing application. Staff can propose changes to the amount a department may spend without additional review. A separate authorized person must approve each change. That is a business rule, not merely a screen layout: the application must prevent self-approval even when someone uses another account, another interface, or an automated integration.

The security objective is not to assume employees are dishonest. It is to limit what one mistaken, misused, or compromised identity can accomplish. A control that depends on everyone remembering the policy provides a different level of protection from a control that refuses a prohibited action.

Authentication, permission, and independence are different questions

Authentication establishes the account making a request. Authorization determines what that account may do. An approval rule adds another question: may this person approve this particular request, given who created it and what it changes? OWASP emphasizes server-side authorization and checking permissions on every request; a successful login is not a substitute for those decisions. [2]

For our purchasing example, an account with approval permission is necessary but insufficient. The approver must also be a different person from the requester. Multifactor authentication on both accounts does not resolve that relationship: the same employee can legitimately authenticate to both.

Define the rule precisely. This article implements one requester plus one independent approver. That is a two-person control. It is not a requirement for two additional approvers, and it does not prevent two people from colluding.

How the approval bypass works

Imagine a fictitious business where Alex has an everyday account and a separate approval account retained from temporary coverage. The accounts have different usernames but belong to the same employee. The purchasing application checks that the usernames differ and that the second account has the approver role.

Different accounts do not establish independent review

Account	Canonical person	Permission
alex.request@example.test	person-1001	Submit a change
alex.approve@example.test	person-1001	Approve a change
jordan.approve@example.test	person-1002	Approve a change

Here, a canonical person identifier means an organization-controlled record linking accounts to the individual assigned to them. It is not a display name or a value supplied in a browser request. The example uses short fictitious identifiers for readability.

- 1. Propose:** Alex requests an increase in the department's purchasing limit from \$500 to \$5,000.
- 2. Switch accounts:** Alex signs in with the approval account.
- 3. Pass the incomplete check:** The server sees a different username with approval permission.
- 4. Apply:** The purchasing limit changes, and the log records a requester and an approver even though no independent person participated.

The defect is a mismatch between the policy and the software's definition of independence. Comparing account identifiers answers whether two accounts differ. It does not answer whether two people participated.

There is a related failure when software allows an action to skip a required approval step altogether. MITRE describes improper workflow enforcement as failing to ensure that required behaviors occur in the intended sequence. The attack can use otherwise valid application functions rather than malformed input. [3]

Other ways the same rule can fail

A visible approval process can also be undermined when the server trusts a submitted approver identifier, an import path updates the record directly, or a background account has unrestricted authority to mark requests approved. Hiding a button does not close those paths. The rule belongs at the server-side operation that changes business state. [2]

Even a genuinely independent approval can become ineffective when the proposed amount changes afterward or an old approval overwrites a newer decision. OWASP's transaction-authorization guidance calls for review of significant transaction details, protection against changes after authorization, and a final authorization check tied to execution. [4]

See the faulty check in code

Local, authorized testing only. The examples use fictitious accounts and local data. The tests create temporary SQLite databases; they do not contact a purchasing system, change live permissions, or send payments. Do not use real purchasing limits or employee accounts to test a production workflow without explicit authorization and a controlled plan.

This complete executable example, `unsafe_check.py`, evaluates the defective account-comparison rule. It is intentionally unsafe and should not be used as an application authorization function.

Python · `unsafe_check.py` — different accounts mistaken for different people

```
ACCOUNTS = {
    "alex.request@example.test": {"person_id": "person-1001", "can_approve": False},
    "alex.approve@example.test": {"person_id": "person-1001", "can_approve": True},
}

def vulnerable_can_approve(maker_account: str, checker_account: str) -> bool:
    checker = ACCOUNTS.get(checker_account)
    return bool(
        checker
        and checker["can_approve"]
        and maker_account != checker_account
    )

if __name__ == "__main__":
    maker = "alex.request@example.test"
    checker = "alex.approve@example.test"
    print("Different accounts:", maker != checker)
    print("Same person:", ACCOUNTS[maker]["person_id"] == ACCOUNTS[checker]["person_id"])
    print("Vulnerable decision:", "ALLOW" if vulnerable_can_approve(maker, checker) else "DENY")
```

Running it produces:

Plaintext · output from the intentionally faulty check

```
Different accounts: True
Same person: True
Vulnerable decision: ALLOW
```

The function has done exactly what its condition says. The problem is that the condition does not express the business requirement. Replacing the usernames with randomly generated account IDs would not fix it; the IDs would still identify accounts rather than independent people.

Also notice what the example does not require: stealing another person's password or breaking encryption. Legitimate access to two accounts is enough under this flawed rule. An external intruder who gains equivalent access could exploit the same configuration.

Build an enforceable approval boundary

A useful repair must address identity, business state, and timing together. The following implementation is a complete local component for requesting and applying purchasing-limit changes. It persists the

policy change and its approval record in SQLite. It is not a login service, purchasing application, or payment connector.

The example adopts specific policy choices: every limit change requires independent approval, requests expire after 15 minutes, amounts use integer US cents, and the largest permitted limit is \$100,000. Those values are examples to be replaced through an organization's approved policy process, not universal security recommendations.

Define the trust boundary before integration

The `actor_id` argument must come from the application's verified login session, not from submitted JSON, a query parameter, or an unverified header. Account-to-person mappings and permissions must come from controlled identity administration. The code reads them from its local authorization catalog; it cannot prove that a poorly maintained catalog reflects reality.

This component intentionally applies to one organization whose listed requesters and approvers have authority over all budgets in the database. A multi-tenant or department-scoped application needs additional organization and resource checks. The example must not be treated as providing those checks implicitly.

The approval interface must display the budget, the current and proposed amounts, the requester, and the request's expiry. It submits the immutable request identifier together with the reviewed budget identifier, version, and proposed amount. The server compares that snapshot with its stored proposal instead of accepting a new amount at approval time. This applies OWASP's guidance to bind authorization to the significant details of an operation. [4]

The complete local control

Save this file as `approval_guard.py`. It requires Python 3.12 or newer and uses Python's standard SQLite interface. The connection uses explicit SQL transactions, and values are passed through parameter binding rather than SQL string interpolation. [5]

Python · `approval_guard.py` — independent approval with an atomic policy update

```
from __future__ import annotations

import sqlite3
import time
import uuid
from contextlib import contextmanager
from collections.abc import Iterator

APPROVAL_TTL_SECONDS = 900
MAX_LIMIT_CENTS = 10_000_000

SCHEMA = """
CREATE TABLE IF NOT EXISTS accounts (
    account_id TEXT PRIMARY KEY,
    person_id TEXT NOT NULL,
    can_request INTEGER NOT NULL CHECK (can_request IN (0, 1)),
    can_approve INTEGER NOT NULL CHECK (can_approve IN (0, 1)),
    active INTEGER NOT NULL CHECK (active IN (0, 1))
);
CREATE TABLE IF NOT EXISTS budgets (
    budget_id TEXT PRIMARY KEY,
    limit_cents INTEGER NOT NULL CHECK (limit_cents >= 0),
    version INTEGER NOT NULL CHECK (version >= 0)
);
```

```

CREATE TABLE IF NOT EXISTS changes (
    change_id TEXT PRIMARY KEY,
    budget_id TEXT NOT NULL REFERENCES budgets(budget_id),
    new_limit_cents INTEGER NOT NULL CHECK (new_limit_cents >= 0),
    base_version INTEGER NOT NULL,
    maker_account TEXT NOT NULL REFERENCES accounts(account_id),
    maker_person TEXT NOT NULL,
    expires_at INTEGER NOT NULL,
    checker_account TEXT REFERENCES accounts(account_id),
    checker_person TEXT,
    applied_at INTEGER,
    CHECK (
        (applied_at IS NULL AND checker_account IS NULL
         AND checker_person IS NULL)
        OR
        (applied_at IS NOT NULL AND checker_account IS NOT NULL
         AND checker_person IS NOT NULL AND checker_person != maker_person)
    )
);
CREATE TRIGGER IF NOT EXISTS immutable_proposal
BEFORE UPDATE OF change_id, budget_id, new_limit_cents, base_version,
                maker_account, maker_person, expires_at ON changes
BEGIN
    SELECT RAISE(ABORT, 'Proposal is immutable; submit a new request');
END;
"""

def open_database(path: str) -> sqlite3.Connection:
    db = sqlite3.connect(path, timeout=5, autocommit=True)
    db.row_factory = sqlite3.Row
    try:
        db.execute("PRAGMA foreign_keys = ON")
        db.executescript(SCHEMA)
    except BaseException:
        db.close()
        raise
    return db

@contextmanager
def write_transaction(db: sqlite3.Connection) -> Iterator[None]:
    db.execute("BEGIN IMMEDIATE")
    try:
        yield
        db.execute("COMMIT")
    except BaseException:
        if db.in_transaction:
            db.execute("ROLLBACK")
        raise

def account(db: sqlite3.Connection, account_id: str) -> sqlite3.Row:
    row = db.execute(
        "SELECT * FROM accounts WHERE account_id = ? AND active = 1",
        (account_id,)
    ).fetchone()
    if row is None:
        raise PermissionError("Unknown or inactive account")
    return row

def submit_change(
    db: sqlite3.Connection, actor_id: str, budget_id: str, limit_cents: int
) -> str:
    if type(limit_cents) is not int or not 0 <= limit_cents <= MAX_LIMIT_CENTS:
        raise ValueError("Limit is outside the configured policy")
    with write_transaction(db):
        maker = account(db, actor_id)
        if not maker["can_request"]:
            raise PermissionError("Request permission required")
        budget = db.execute(
            "SELECT * FROM budgets WHERE budget_id = ?", (budget_id,)
        ).fetchone()

```

```

    if budget is None:
        raise ValueError("Unknown budget")
    if budget["limit_cents"] == limit_cents:
        raise ValueError("No change requested")
    change_id = str(uuid.uuid4())
    db.execute(
        "INSERT INTO changes "
        "(change_id, budget_id, new_limit_cents, base_version, "
        "maker_account, maker_person, expires_at) VALUES (?, ?, ?, ?, ?, ?, ?)",
        (change_id, budget_id, limit_cents, budget["version"], actor_id,
         maker["person_id"], int(time.time()) + APPROVAL_TTL_SECONDS),
    )
    return change_id

def approve_and_apply(
    db: sqlite3.Connection, actor_id: str, change_id: str,
    reviewed_budget_id: str, reviewed_version: int, reviewed_limit_cents: int,
) -> None:
    with write_transaction(db):
        checker = account(db, actor_id)
        if not checker["can_approve"]:
            raise PermissionError("Approval permission required")
        change = db.execute(
            "SELECT * FROM changes WHERE change_id = ?", (change_id,)
        ).fetchone()
        if change is None:
            raise ValueError("Unknown change")
        maker = account(db, change["maker_account"])
        if not maker["can_request"] or maker["person_id"] != change["maker_person"]:
            raise PermissionError("Requester authority has changed")
        if checker["person_id"] == change["maker_person"]:
            raise PermissionError("An independent person must approve")
        now = int(time.time())
        if change["applied_at"] is not None or now >= change["expires_at"]:
            raise ValueError("Change is already applied or expired")
        if (type(reviewed_version) is not int
            or type(reviewed_limit_cents) is not int
            or (reviewed_budget_id, reviewed_version, reviewed_limit_cents)
            != (change["budget_id"], change["base_version"],
                change["new_limit_cents"])):
            raise ValueError("Reviewed details do not match")
        if not 0 <= change["new_limit_cents"] <= MAX_LIMIT_CENTS:
            raise ValueError("Change no longer satisfies policy")
        result = db.execute(
            "UPDATE budgets SET limit_cents = ?, version = version + 1 "
            "WHERE budget_id = ? AND version = ?",
            (change["new_limit_cents"], change["budget_id"], change["base_version"]),
        )
        if result.rowcount != 1:
            raise ValueError("Budget changed; submit a fresh request")
        db.execute(
            "UPDATE changes SET checker_account = ?, checker_person = ?, "
            "applied_at = ? WHERE change_id = ?",
            (actor_id, checker["person_id"], now, change_id),
        )

```

Why these checks work together

The request captures the requester's person identifier when it is created. At approval, the code checks both current authority and that historical identity. Merely relabeling the original account in the catalog cannot silently make the existing request eligible for self-approval: the changed mapping causes rejection.

The approver is compared with the stored requester person, not just the requester account. Consequently, Alex's second account is rejected while Jordan's independently assigned account can

approve. This depends on accurate account ownership records; it does not establish who is physically operating a stolen session.

The proposal fields cannot be edited through ordinary SQL updates while the trigger remains in place. A changed amount or target requires a new request. Separately, the budget version stops an older pending request from overwriting a more recent approved change. These are different protections: immutability protects what was proposed, while version checking protects the state to which it applies.

`BEGIN IMMEDIATE` starts a SQLite write transaction before the checks and updates. SQLite permits only one simultaneous writer. The budget update and approval record therefore commit together, or the transaction rolls back. Lock contention can still raise a database error; it must never be interpreted as permission to proceed without the checks. [6]

Once a request has an `applied_at` value, another attempt is rejected. The component deliberately treats retries as a state to investigate rather than applying the change twice. A user interface should refresh the recorded outcome after an uncertain response instead of creating a replacement request automatically.

The database is inside the trust boundary. Someone who can rewrite the database file, change identity mappings, remove triggers, or replace the application code can bypass this component. Its decision records are not tamper-proof. Restrict those administrative capabilities and preserve evidence outside the same administrator's unrestricted control.

Test the control, not just the button

A successful approval proves only that an allowed path works. The important security evidence is that forbidden paths fail without changing business state. OWASP's workflow-testing guidance recommends application-specific misuse cases based on the required sequence and business rules. [8]

The following complete test file checks same-person aliases, valid independent approval, replay, stale requests, expiry, altered review details, revoked approval permission, and attempted proposal edits. Account rows are test fixtures, not a substitute for production authentication.

Python · `test_approval_guard.py` — eight executable approval-control tests

```
import sqlite3
import tempfile
import unittest
from pathlib import Path
from unittest.mock import patch

from approval_guard import open_database, submit_change, approve_and_apply

MAKER = "alex.request@example.test"
ALIAS = "alex.approve@example.test"
CHECKER = "jordan.approve@example.test"

class ApprovalTests(unittest.TestCase):
    def setUp(self):
        self.tmp = tempfile.TemporaryDirectory()
        self.db = open_database(str(Path(self.tmp.name) / "approval.db"))
        self.db.executemany(
            "INSERT INTO accounts VALUES (?, ?, ?, ?, ?)",
            [(MAKER, "person-1001", 1, 0, 1),
             (ALIAS, "person-1001", 0, 1, 1),
```

```

        (CHECKER, "person-1002", 0, 1, 1)],
    )
    self.db.execute("INSERT INTO budgets VALUES ('purchasing', 50000, 0)")

def tearDown(self):
    self.db.close()
    self.tmp.cleanup()

def request(self, amount=500000):
    return submit_change(self.db, MAKER, "purchasing", amount)

def approve(self, actor, change_id, amount=500000):
    approve_and_apply(self.db, actor, change_id, "purchasing", 0, amount)

def test_same_person_second_account_is_denied(self):
    change_id = self.request()
    with self.assertRaises(PermissionError):
        self.approve(ALIAS, change_id)
    self.assertEqual(self.db.execute(
        "SELECT limit_cents FROM budgets"
    ).fetchone()[0], 50000)

def test_independent_approval_updates_budget_and_record(self):
    change_id = self.request()
    self.approve(CHECKER, change_id)
    row = self.db.execute("SELECT * FROM budgets").fetchone()
    self.assertEqual((row["limit_cents"], row["version"]), (500000, 1))
    decision = self.db.execute("SELECT * FROM changes").fetchone()
    self.assertEqual(decision["checker_person"], "person-1002")
    self.assertIsNotNone(decision["applied_at"])

def test_replay_is_denied(self):
    change_id = self.request()
    self.approve(CHECKER, change_id)
    with self.assertRaises(ValueError):
        self.approve(CHECKER, change_id)

def test_stale_request_cannot_overwrite_newer_decision(self):
    first = self.request()
    second = self.request(700000)
    self.approve(CHECKER, second, 700000)
    with self.assertRaises(ValueError):
        self.approve(CHECKER, first)

def test_expired_request_is_denied(self):
    with patch("approval_guard.time.time", return_value=1000):
        change_id = self.request()
    with patch("approval_guard.time.time", return_value=1900):
        with self.assertRaises(ValueError):
            self.approve(CHECKER, change_id)

def test_reviewed_amount_must_match(self):
    with self.assertRaises(ValueError):
        self.approve(CHECKER, self.request(), 900000)

def test_revoked_approver_is_denied(self):
    change_id = self.request()
    self.db.execute(
        "UPDATE accounts SET can_approve = 0 WHERE account_id = ?", (CHECKER,)
    )
    with self.assertRaises(PermissionError):
        self.approve(CHECKER, change_id)

def test_proposal_fields_cannot_be_edited(self):
    change_id = self.request()
    with self.assertRaises(sqlite3.IntegrityError):
        self.db.execute(
            "UPDATE changes SET new_limit_cents = 900000 WHERE change_id = ?",
            (change_id,)
        )

```

```
if __name__ == "__main__":  
    unittest.main()
```

On Ubuntu with Python 3.12 or newer, place `approval_guard.py` and `test_approval_guard.py` in the same working directory, then run:

Bash · run the local approval-control tests

```
python3 -m unittest -v test_approval_guard.py
```

All eight tests should pass. In particular, the self-approval test verifies that the budget remains at \$500 after Alex's alternate account is rejected. The independent-approval test verifies both the \$5,000 limit and the recorded independent reviewer.

A useful acceptance criterion: An authorized, independent reviewer can apply the exact pending change once. A same-person alias, expired request, changed review snapshot, or stale version cannot apply it.

Before integrating the component, extend the tests to the real application's authentication, tenant boundaries, administrator routes, imports, scheduled jobs, and error handling. Include concurrent approval attempts and failure while writing the decision record. Local tests of the component do not establish that every surrounding application path reaches it.

Controls the code cannot replace

Protect the people and permission records

Assign responsibility for linking all employee accounts to the correct person. Review alternate accounts, inherited groups, delegated roles, and administrative access as one person's combined authority. Make changes to that catalog independently reviewable; otherwise, someone can evade the rule by changing who the system believes owns an account.

For connected identity systems, define how quickly permission removals reach this catalog and what happens when synchronization fails. The code checks its database's current records, not the instantaneous state of an external identity provider. That distinction belongs in the risk assessment and operating procedures.

Make review meaningful and available

A reviewer needs context, not just an approval button: what changes, why it is needed, who benefits, what evidence supports it, and what authority the requester already holds. For unusually consequential changes, require fresh authentication through the existing identity system before approval.

Plan reviewer coverage before an urgent request arrives. A permanent self-approval override introduced for convenience defeats the purpose of the control. For a very small business, an appropriately authorized external reviewer, narrower limits, delayed execution, or independent reconciliation may help manage risk. Those alternatives are not automatically equivalent to preventive two-person approval; document the remaining exposure rather than relabeling it as solved.

Preserve evidence without collecting secrets

Record the request, requester and reviewer identities, policy version, reviewed values, timestamps, outcome, and a correlation identifier. Log rejected attempts and authorization changes as well as successes. Protect logs against unauthorized modification, and avoid placing passwords, session tokens, or unnecessary sensitive data in them. OWASP's logging guidance supports recording authorization failures and attributable application events while excluding secrets. [7]

The example stores successful decisions in `changes`; rejected transactions roll back. Production denial monitoring therefore belongs outside that rolled-back transaction. Establish an owner who reviews alerts and can distinguish a mistaken retry from a repeated attempt to evade policy.

Respect the boundary of a local transaction

The component changes a purchasing policy record. It does not enforce purchase approval at checkout, contact a bank, or release money. The purchasing system must actually consult the authoritative policy and enforce its own transaction rules. Saving an approved configuration that downstream software ignores provides no spending protection.

Likewise, a commit in this SQLite database does not make a later external action part of the same transaction. A real integration needs its own failure handling, duplicate-operation controls, and reconciliation. Keep that work explicit rather than assuming that an approval record proves an external action completed.

Potential business repercussions

The following are possible consequences of the constructed scenario, not reported losses or predictions about any particular business. The impact depends on what the workflow controls and what other safeguards remain.

- **Unauthorized commitments:** A self-approved spending increase could let orders proceed without the review management intended. Four unauthorized \$4,500 orders would create \$18,000 in commitments; that arithmetic is an illustration, not a loss estimate.
- **Operational interruption:** Discovery may require suspending changes, reviewing affected orders, correcting limits, and reconciling activity before normal processing resumes. Staff time spent rebuilding trustworthy records is part of the impact.
- **Misleading management evidence:** Reports showing separate requester and approver usernames could overstate the effectiveness of review. Leadership may make decisions on an incorrect picture of how authority was exercised.
- **Wider administrative exposure:** Applied to privileged-access or infrastructure-change workflows, the same independence defect could allow one person to request and authorize a consequential configuration change. The affected privileges determine the potential damage.
- **Customer, supplier, and employee trust:** Incorrect orders or disputed decisions can strain relationships. Weak identity records can also make it harder to distinguish individual misconduct from a system that allowed an inappropriate action.

A useful assessment asks not just whether self-approval is possible, but how much authority it unlocks, how many actions could occur before detection, and whether the organization can reconstruct and reverse those actions.

A practical Sunimod project: an Approval Integrity Review

This is a solvable business project: identify where independent decisions are required, locate the paths that can bypass them, and implement controls that match the actual workflow. An organization does not necessarily need a new enterprise platform. The starting point may be a custom portal, an internal application, or an existing system's supported permissions and integration features.

Hire Sunimod to scope an Approval Integrity Review around a specific outcome, such as preventing self-approved purchasing-limit changes. The proposed work should connect management's policy to application behavior and deliver evidence that the agreed controls operate as intended.

Map authority before changing software

Identify the process owner, the protected business outcome, and the maximum consequence of an unauthorized change. Map requesters, approvers, role administrators, service accounts, and people who can alter the underlying records. Include the people behind alternate accounts, not just the account names visible in a report.

Implement the smallest complete control

Agree on which actions need independent review and which can remain low-friction. Then address server-side checks, immutable proposals, current permissions, expiry, concurrent changes, and evidence. Remove or constrain alternate update paths so that an import, administrator screen, or integration does not bypass the repaired workflow.

Finish with evidence and ownership

- Each protected workflow has a named business owner and a written independence rule.
- Relevant accounts are linked to controlled person identities, with documented exceptions.
- Approval binds to the exact action and cannot be reused for a different or outdated change.
- Negative tests cover self-approval, unauthorized roles, alternate paths, expiry, and concurrent activity.
- Decision evidence and denial monitoring have an assigned reviewer and an appropriate retention policy.
- Reviewer absence and emergency exceptions have a defined process rather than a permanent bypass.

Measure the results using evidence: conflicting account assignments found and resolved, protected update paths verified, exceptions still open, and misuse tests passing. Approval speed also matters; a control that repeatedly blocks legitimate operations needs a better coverage process, not an undocumented escape hatch.

To discuss an existing workflow or a new internal tool, [request an approval-workflow project from Sunimod](#). Describe the system, who proposes changes, who should approve them, and the business consequence you need to prevent.

The takeaway

An approval badge is not proof of independent authority. The meaningful questions are who the people are, what each is allowed to do, exactly what was reviewed, and whether those conditions still hold when the change takes effect.

Start with one consequential workflow. Make its rule explicit, enforce it at the point where business state changes, test the prohibited paths, and assign someone to keep the control accurate as accounts and responsibilities evolve.

Sources and further reading

1. [NIST: Separation of Duty](#) — the distinction between static and dynamic separation.
2. [OWASP: Authorization Cheat Sheet](#) — server-side authorization and permission checks.
3. [MITRE: CWE-841, Improper Enforcement of Behavioral Workflow](#) — missing or bypassed workflow requirements.
4. [OWASP: Transaction Authorization Cheat Sheet](#) — significant transaction details and final execution checks.
5. [Python 3.12: sqlite3 documentation](#) — connections, parameter binding, and transaction control.
6. [SQLite: Transaction documentation](#) — write transactions, contention, commit, and rollback.
7. [OWASP: Logging Cheat Sheet](#) — attributable security events and protection of sensitive information.
8. [OWASP: Testing for the Circumvention of Work Flows](#) — misuse cases tailored to the application's business rules.

The purchasing scenario, policy values, account records, and code examples are original instructional examples. They do not describe an incident at Sunimod or identify a vulnerability in a named commercial product.

Make independent approval a control your business can verify

Bring Sunimod the workflow that protects your spending, access, or operational changes. Hire us to review its approval boundaries, implement an agreed solution, and test the conditions that should never be allowed.

[Request an approval-workflow quote](#)

Describe the business problem in the form. Do not include passwords, access tokens, or sensitive transaction details.