CONTROL TOWER FOR CLAUDE · GUIDE

HIPAA audit logs for Claude Enterprise: check that logging stayed on and every review was signed within your own limit

Daily logging, activity reviews and leaver access, checked from Claude Activity Feed records with a DuckDB script

Published 2026-10-31 · Online version and downloads: www.datameaning.com/guides/hipaa-audit-logs-for-claude-enterprise/

HIPAA's Security Rule asks you to record activity in systems with patient data (164.312(b)) and regularly review those records (164.308(a)(1)(ii)(D)). The text sets no review frequency, so pick one you can justify. The script marks each day recorded or not, and each review in your log signed within your limit or not. This is our reading, not an official list.

What you will need

- The Compliance API turned on for your organization, the day you first turned it on, and a key that can read the Activity Feed: a Compliance Access Key or an Admin API key with the `read:compliance_activities` scope (Anthropic: Query the Activity Feed (read 2026-10-31)).
- Your activity review log as a CSV: the first and last day each review covered, the reviewer, and the day the reviewer signed, every date written YYYY-MM-DD.
- The roster of people your procedure allows to sign a review.
- An HR leaver list: each person's work email and last day of work, the day written YYYY-MM-DD.
- Python 3 and the `duckdb` module.

Step 1. Pull the feed for the whole period, once, as JSON lines

The Activity Feed returns each activity with an `id`, a `created_at`, an `actor` and a `type` (Anthropic: the Activity object (read 2026-10-31)). Activities are queryable within 1 minute and retained for 6 years (Anthropic: Query the Activity Feed (read 2026-10-31)). Anthropic treats the feed as at-least-once, so a retry can deliver a record twice, and it tells you to deduplicate on `id`. To show a run is complete, it tells you to log the starting cursor, the final `last_id`, the number of records, the run time and the request-id of the final page (Anthropic: Delivery guarantees and completeness (read 2026-10-31)). Keep those with the pull.

The script reads a file; it does not call the API. We did not call the real API, because we have no organization to call it on. Save the pull as one JSON object per line, in the shape Anthropic documents.

Step 2. Load the feed, the review log and the roster into DuckDB

DuckDB's `read_json` takes `format='newline_delimited'` and a `columns` list; missing keys become NULL and keys you do not list are left out (DuckDB: loading JSON (read 2026-10-31)). A field your pull lacks does not stop the load. The review log and the HR list are read as text, and every date in them is read only as YYYY-MM-DD, because DuckDB's own date cast is lenient and reads 8/5/26 as the year 8. So a period date, signature date or last day not written as YYYY-MM-DD, such as 8/5/26 or 09/24/2026, becomes NULL instead of moving the date or crashing the run, and an input check counts it. Emails are lower-cased and trimmed, so `Priya.Shah@Corp.Example` in the log and `priya.shah@corp.example` on the roster are the same person.

```
con.execute(f"""CREATE TABLE reviews_raw AS SELECT
          {strict_date(REVIEW_START_COL)} AS period_start, {strict_date(REVIEW_END_COL)} AS
period_end,
          lower(trim("{REVIEWER_COL}")) AS reviewer, {strict_date(SIGNED_COL)} AS signed_on,
          trim(coalesce("{SIGNED_COL}", '')) AS signed_text
          FROM read_csv(?, header=true, all_varchar=true)""", [rvw])
```

Removal records name the removed user in different ways. Anthropic's API reference gives `org_user_deleted` a `deleted_user_email` and a `deleted_user_id`, and gives `scim_user_deleted` only a `user_id` (Anthropic: Query compliance activities, API reference (read 2026-10-31)). The script reads the email from `TARGET_FIELD`, and when a record carries only an id it looks the email up through `actor.user_id` on that person's other activity. The reference does not say these ids are the same, so that part is our assumption. Open one of your own removal records to check.

Step 3. Run nine input checks before building anything

A list built from an incomplete feed is a risky kind of evidence, because it looks complete. The script runs nine checks first, and each one names the planted fault that trips it:

- **Duplicate activity ids** (a duplicated record).
- **Records with no id or no readable `created_at`** (an unreadable timestamp).
- **Days on or after enablement with no records** (five days with no records).
- **Days in the period before the Compliance API was enabled** (the Compliance API turned on late).
- **Review log rows with no readable period start or end** (a review row with no end date).
- **Review log rows with a signature date that is not YYYY-MM-DD** (a signature date written 09/24/2026).
- **Reviewer roster rows with no email** (a NULL in the roster).
- **HR leaver rows with no email or no last day** (a NULL in the HR file).
- **HR leaver rows whose last day is not written as YYYY-MM-DD** (a last day written 8/5/26).

If any fails, all four lists are printed with `INCOMPLETE` in their heading. The script flags duplicate ids but does not remove them: remove them before you run it, keeping the first record for each id.

Step 4. 164.312(b): was the feed recording on every day?

The script builds one row for each day of the period with the number of records dated that day. A day with records is `recorded`. A day with none that falls before `COMPLIANCE_API_ENABLED_ON` is `BEFORE ENABLED`: Anthropic does not backfill, so you cannot evidence that day from the feed, and the honest answer is to say so. Any other day with none is `NO RECORDS`.

```
days = con.execute(f"""
    SELECT d::DATE AS day, count(a.id) AS records,
           CASE WHEN count(a.id) > 0 THEN 'recorded'
                WHEN d::DATE < {en} THEN 'BEFORE ENABLED' ELSE 'NO RECORDS' END AS status
    FROM generate_series(DATE '{p0}', DATE '{p1}', INTERVAL 1 DAY) t(d)
    LEFT JOIN activities a ON a.ts::DATE = d::DATE GROUP BY 1 ORDER BY 1""").fetchall()
```

Anthropic's FAQ says that for a Claude Console organization, activity not recorded while the Compliance API is off cannot be recovered, and that turning it off or on there is recorded as an `org_compliance_api_settings_updated` activity (Anthropic: Compliance API FAQ (read 2026-10-31)). The API reference lists that type without limiting it to Console organizations ("Organization compliance API settings were updated", Anthropic: Query compliance activities, API reference (read

2026-10-31)). So for a `NO RECORDS` stretch, search the feed for that type around its edges, and ask who changed the setting and why.

One caution: Anthropic says activity volume is not a completeness check, because the number of `_viewed` events depends on how each app loads content (Anthropic: Delivery guarantees and completeness (read 2026-10-31)). A day with no records of any kind is a different signal, but a small organization can have a quiet day, and a day with some records but not all is invisible to this check. Treat a failure as a question to answer, not as proof of a gap.

Step 5. 164.308(a)(1)(ii)(D): every review, signed within your limit by someone on the roster

The review log is your own record, so the script checks it against the settings you wrote down. Each review that overlaps the period gets one status, the first that applies: `SIGNATURE DATE UNREADABLE` (a signature date not written as YYYY-MM-DD, such as 8/5/26 or 09/24/2026, also counted by an input check), `NOT SIGNED` (no signature date), `SIGNED BEFORE PERIOD ENDED` (signed before the last day it covers), `NO REVIEWER`, `NOT ON ROSTER`, `LATE` (signed more than `SIGN_WITHIN_DAYS` days after its last day) or `on time`. We set `SIGN_WITHIN_DAYS` to 14 as an example. The Security Rule text we quote gives no number, so use the one in your procedure.

```
    sql = f"""
        SELECT v.period_start, v.period_end, coalesce(v.reviewer, '-') AS reviewer, v.signed_on,
               date_diff('day', v.period_end, v.signed_on) AS days_to_sign,
               CASE WHEN v.signed_on IS NULL AND v.signed_text <> '' THEN 'SIGNATURE DATE
UNREADABLE'
                    WHEN v.signed_on IS NULL THEN 'NOT SIGNED'
                    WHEN v.signed_on < v.period_end THEN 'SIGNED BEFORE PERIOD ENDED'
                    WHEN v.reviewer IS NULL OR v.reviewer = '' THEN 'NO REVIEWER'
                    WHEN v.reviewer NOT IN (SELECT email FROM reviewers WHERE email IS NOT NULL)
THEN 'NOT ON ROSTER'
                    WHEN date_diff('day', v.period_end, v.signed_on) > {SIGN_WITHIN_DAYS} THEN
'LATE'
                    ELSE 'on time' END AS status
        FROM reviews v WHERE v.period_start <= DATE '{p1}' AND v.period_end >= DATE '{p0}'
        ORDER BY v.period_start, v.period_end, v.reviewer"""
```

Step 6. Find the days no review covers

A status on each row cannot show a review that was never written down. So the script also lists every stretch of days in the period that no readable review row covers. It uses a gaps-and-islands query: days with no covering row, grouped into runs of consecutive days.

```
        WITH d AS (SELECT d::DATE AS day FROM generate_series(DATE '{p0}', DATE '{p1}', INTERVAL 1
DAY) t(d)
                   WHERE NOT EXISTS (SELECT 1 FROM reviews v WHERE v.period_start <= d::DATE AND
v.period_end >= d::DATE)),
```

```
            g AS (SELECT day, day - (row_number() OVER (ORDER BY day))::INTEGER AS grp FROM d)
        SELECT min(day) AS first_day, max(day) AS last_day, count(*) AS days FROM g GROUP BY grp
ORDER BY 1"""
```

A review row with an unreadable date is not used, so its days appear here too, next to the input check that counted the row. The list counts days covered by any readable row; the status list above says whether each of those rows was signed in time by someone on the roster.

Step 7. 164.308(a)(3)(ii)(C): leavers, and a fingerprint on each list

The leaver list uses the same logic as our SOC 2 guide for Claude Enterprise, where it is explained in full. It starts from the HR list, credits each removal record to one departure, and marks a leaver `NO REMOVAL RECORD`, `REMOVED LATE` or `ACTIVE AFTER REMOVAL` when the feed shows it. Without a seat list, a leaver with no Claude activity seen reads `CHECK: NO CLAUDE ACTIVITY SEEN`; clear those against your seat list. Whether a removal that late is acceptable under your termination procedure is for you to decide, because the text above gives no number.

Each list is printed with its period, filters, run time, row count and the SHA-256 below, computed over every row of the list in the order written to the CSV:

```
def fingerprint(rows):
    body = "\n".join("|".join("" if v is None else str(v) for v in r) for r in rows)
    return hashlib.sha256(body.encode()).hexdigest()
```

The fingerprint does not prove the rows are right. It lets anyone who re-runs the script on the same file confirm they got the same rows, and it shows if a list changed after it was handed over.

The code

The complete download, `hipaa_claude_review_evidence.py`, as run. The same file is at https://www.datameaning.com/guides/hipaa-audit-logs-for-claude-enterprise/hipaa_claude_review_evidence.py.

```
#!/usr/bin/env python3
"""HIPAA audit-log evidence for Claude Enterprise from Compliance API Activity Feed records, in
DuckDB.

    python3 hipaa_claude_review_evidence.py            # clean generated records
    python3 hipaa_claude_review_evidence.py --plant    # generated records with planted faults
    python3 hipaa_claude_review_evidence.py --strict   # exit 1 when anything needs follow-up

Builds four lists, each printed with its period, filters, run time, row count and a SHA-256 of its
rows:
  164.312(b)             one row per day of the period: how many Activity Feed records exist, and
whether the
                         day is recorded, has NO RECORDS, or falls BEFORE ENABLED (the Compliance
```

```
API was not on yet)
  164.308(a)(1)(ii)(D)   every review in your review log that overlaps the period, with its
reviewer, the date
                         signed, the days from the end of the period reviewed to the signature, and
a status
  164.308(a)(1)(ii)(D)   every stretch of days in the period that no review row covers
  164.308(a)(3)(ii)(C)   every leaver in the period, the date of the first removal record credited
to that
                         departure, the days between, and any use of the account after that removal
Before it builds them it runs nine input checks and marks the lists INCOMPLETE if one fails.

The generated records are NOT a real organization's data. Field names (id, created_at, actor.type,
actor.email_address, actor.user_id, type) come from Anthropic's Activity object:
https://platform.claude.com/docs/en/manage-claude/compliance-activity-feed
The activity-type strings and the removal fields come from the API reference:
https://platform.claude.com/docs/en/api/compliance/activities/list
There, org_user_deleted names the removed user in deleted_user_email and deleted_user_id, and
scim_user_deleted carries only user_id. The script reads the email from TARGET_FIELD, and maps the
ids
to an email through actor.user_id seen elsewhere in the feed. The reference does not say that these
ids are the same as actor.user_id; we assume so. Check one of your own removal records.
"""
import csv
import datetime as dt
import hashlib
import json
import pathlib
import random
import sys

import duckdb

# ---- CHANGE THESE FOR YOUR OWN DATA ------------------------------------------------------
PERIOD_START = "2026-07-06"          # first day of the period you are evidencing (UTC, inclusive)
PERIOD_END = "2026-09-27"            # last day of the period (UTC, inclusive)
COMPLIANCE_API_ENABLED_ON = None     # the day you first turned the Compliance API on, YYYY-MM-DD;
None = not known
ACTIVITIES_FILE = None               # your Activity Feed pull, one JSON object per line; None =
generate
REVIEWS_FILE = None                  # your activity review log (CSV with a header row); None =
generate
REVIEWERS_FILE = None                # your roster of people allowed to sign a review (CSV); None =
generate
LEAVERS_FILE = None                  # your HR leaver list (CSV with a header row); None = generate
REVIEW_START_COL = "period_start"    # first day the review covered, as YYYY-MM-DD
REVIEW_END_COL = "period_end"        # last day the review covered, as YYYY-MM-DD
REVIEWER_COL = "reviewer"            # the reviewer's work email
SIGNED_COL = "signed_on"             # the day the reviewer signed, as YYYY-MM-DD; empty = not
signed
REVIEWER_EMAIL_COL = "email"         # on the roster
LEAVER_EMAIL_COL = "email"
LEAVER_DATE_COL = "last_day"         # the leaver's last day of work, as YYYY-MM-DD
TARGET_FIELD = "deleted_user_email"  # the field on a removal record that holds the removed user's
email
SIGN_WITHIN_DAYS = 14                # a review must be signed within this many days after the
period it covers ends
GRACE_DAYS = 1                       # access must end within this many days of the last day of
work
```

```
REMOVAL_WINDOW_DAYS = 30            # a removal before every last day counts if it is at most this
many days early
# ---------------------------------------------------------------------------------------------

REMOVAL_TYPES = ["org_user_deleted", "scim_user_deleted", "org_user_left"]
OUT_DIR = pathlib.Path("evidence-out")
GEN_DIR = pathlib.Path("generated")


def generate(plant):
    """Seeded stand-in for an organization's feed, review log, reviewer roster and HR leaver
list."""
    rnd = random.Random(222)
    GEN_DIR.mkdir(exist_ok=True)
    users = [f"u{n:02d}@example.test" for n in range(1, 41)]
    reviewers = ["privacy.officer@example.test", "security.lead@example.test"]
    leavers = users[29:35]
    last_day = {u: dt.date(2026, 7, 10) + dt.timedelta(days=11 * i) for i, u in enumerate(leavers)}
    start, end = dt.date.fromisoformat(PERIOD_START), dt.date.fromisoformat(PERIOD_END)
    recs = []

    def add(day, typ, actor, **extra):
        t = dt.datetime.combine(day, dt.time(rnd.randrange(7, 19), rnd.randrange(60),
rnd.randrange(60)))
        recs.append({"id": "", "created_at": t.strftime("%Y-%m-%dT%H:%M:%SZ"),
                     "organization_id": "org_generated", "actor": actor, "type": typ, **extra})

    def user(email):
        return {"type": "user_actor", "email_address": email, "user_id": "user_" + email.split("@")
[0],
                "ip_address": "192.0.2.1", "user_agent": "generated"}

    leaver_kept, leaver_late, came_back = leavers[3], leavers[4], leavers[0]
    day = start
    while day <= end:
        staff = [u for u in users if u not in last_day or day <= last_day[u]]
        if plant and day > last_day.get(leaver_kept, dt.date.max) and (day -
last_day[leaver_kept]).days <= 10:
            staff.append(leaver_kept)
            add(day, "claude_chat_created", user(leaver_kept))
        add(day, "sso_login_succeeded", user(rnd.choice(staff)))
        for _ in range(rnd.randrange(3, 8)):
            add(day, rnd.choice(["claude_chat_created", "claude_chat_viewed"]),
user(rnd.choice(staff)))
        day += dt.timedelta(days=1)
    for u in leavers:
        if plant and u == leaver_kept:
            continue
        lag = 9 if (plant and u == leaver_late) else rnd.choice([0, 1])
        add(last_day[u] + dt.timedelta(days=lag), "org_user_deleted",
            {"type": "scim_directory_sync_actor", "directory_id": "directory_generated",
             "idp_connection_type": "AzureSCIMV2"},
            **{TARGET_FIELD: u, "deleted_user_id": "user_" + u.split("@")[0]})
    enabled = PERIOD_START
    if plant:
        add(last_day[came_back] + dt.timedelta(days=10), "claude_chat_created", user(came_back))
        enabled = "2026-07-08"                                  # planted: Compliance API
turned on two days late
```

```
        recs = [r for r in recs if r["created_at"][:10] >= enabled
                and not "2026-08-10" <= r["created_at"][:10] <= "2026-08-14"]   # planted: five
days with no records
    recs.sort(key=lambda r: (r["created_at"], r["type"], json.dumps(r["actor"], sort_keys=True)))
    for n, r in enumerate(recs, 1):
        r["id"] = f"activity_{n:06d}"
    if plant:
        recs[60]["created_at"] = "not-a-date"                          # planted: an unreadable
timestamp
        recs.append(dict(recs[40]))                                    # planted: a duplicated
record
    feed = GEN_DIR / ("feed_planted.jsonl" if plant else "feed_clean.jsonl")
    feed.write_text("".join(json.dumps(r) + "\n" for r in recs))

    reviews = []
    for k in range(12):                                                # one review a week, Monday
to Sunday
        s = start + dt.timedelta(days=7 * k)
        e = s + dt.timedelta(days=6)
        signed = e + dt.timedelta(days=rnd.randrange(1, 7))
        who = rnd.choice(reviewers)
        row = [s.isoformat(), e.isoformat(), who, signed.isoformat()]
        if plant and k == 1:
            row[3] = signed.strftime("%m/%d/%Y")                       # planted: a signature date
not written YYYY-MM-DD
        if plant and k == 3:
            continue                                                   # planted: the week nobody
reviewed
        if plant and k == 6:
            row[3] = (e + dt.timedelta(days=33)).isoformat()           # planted: signed 33 days
after the week ended
        if plant and k == 8:
            row[2] = "u11@example.test"                                # planted: signed by someone
not on the roster
        if plant and k == 9:
            row[1] = ""                                                # planted: a review row with
no end date
        if plant and k == 10:
            row[3] = ""                                                # planted: a review nobody
signed
        if plant and k == 11:
            row[3] = (e - dt.timedelta(days=3)).isoformat()            # planted: signed before the
week ended
        reviews.append(row)
    with open(GEN_DIR / "reviews.csv", "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["period_start", "period_end", "reviewer", "signed_on"])
        w.writerows(reviews)
    with open(GEN_DIR / "reviewers.csv", "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["email", "role"])
        w.writerows([[r, "reviewer"] for r in reviewers] + ([["", "alternate"]] if plant else []))
    with open(GEN_DIR / "leavers.csv", "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["email", "last_day"])
        extra = [["", "2026-09-10"],                    # planted: a NULL email in the HR file
                 ["u43@example.test", "8/5/26"]]        # planted: a last day not written as YYYY-
MM-DD
```

```
        w.writerows([[u, last_day[u].isoformat()] for u in leavers] + (extra if plant else []))
    return (str(feed), str(GEN_DIR / "reviews.csv"), str(GEN_DIR / "reviewers.csv"),
            str(GEN_DIR / "leavers.csv"), enabled)


def fingerprint(rows):
    body = "\n".join("|".join("" if v is None else str(v) for v in r) for r in rows)
    return hashlib.sha256(body.encode()).hexdigest()


def population(con, name, control, filters, sql, header, complete):
    rows = con.execute(sql).fetchall()
    OUT_DIR.mkdir(exist_ok=True)
    with open(OUT_DIR / f"{name}.csv", "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(header)
        w.writerows(rows)
    print(f"\n== {control}: {name}" + ("" if complete else "   ** INCOMPLETE: see input checks
**"))
    print(f"period    : {PERIOD_START} to {PERIOD_END} (UTC, inclusive)")
    print(f"filters   : {filters}")
    print(f"run at    : {dt.datetime.now(dt.timezone.utc):%Y-%m-%d %H:%M:%S} UTC")
    print(f"rows      : {len(rows)}")
    print(f"sha256    : {fingerprint(rows)}")
    return rows


def ranges(days):
    """Collapse sorted dates into (first, last, count) stretches."""
    out = []
    for d in days:
        if out and (d - out[-1][1]).days == 1:
            out[-1][1] = d
            out[-1][2] += 1
        else:
            out.append([d, d, 1])
    return out


ISO_DATE = r"\d{4}-\d{2}-\d{2}"


def strict_date(col):
    """A CSV date column read only as YYYY-MM-DD. DuckDB's lenient cast reads 8/5/26 as the year 8,
so
    any other form becomes NULL here and is counted by an input check instead of moving the
date."""
    return (f"CASE WHEN regexp_full_match(trim(\"{col}\"), '{ISO_DATE}') "
            f"THEN try_strptime(trim(\"{col}\"), '%Y-%m-%d')::DATE END")


def main():
    plant, strict = "--plant" in sys.argv, "--strict" in sys.argv
    acts, rvw, rost, lvr, enabled = ACTIVITIES_FILE, REVIEWS_FILE, REVIEWERS_FILE, LEAVERS_FILE,
COMPLIANCE_API_ENABLED_ON
    if acts is None:
        acts, rvw, rost, lvr, enabled = generate(plant)
        print(f"input     : GENERATED records (not a real organization), seed 222, plant={plant}")
```

```
    else:
        missing = [n for n, v in [("REVIEWS_FILE", rvw), ("REVIEWERS_FILE", rost), ("LEAVERS_FILE",
lvr)] if v is None]
        if missing:
            sys.exit("set " + ", ".join(missing) + " as well as ACTIVITIES_FILE")
        print("input     : " + acts)
    print("enabled on: " + (enabled or "not known (COMPLIANCE_API_ENABLED_ON not set)"))
    con = duckdb.connect()
    con.execute(f"""
        CREATE TABLE raw AS SELECT * FROM read_json(?, format='newline_delimited', columns={{
          'id':'VARCHAR', 'created_at':'VARCHAR', 'type':'VARCHAR',
          'actor':'STRUCT(type VARCHAR, email_address VARCHAR, user_id VARCHAR)',
          '{TARGET_FIELD}':'VARCHAR', 'deleted_user_id':'VARCHAR', 'user_id':'VARCHAR'}})""",
[acts])
    con.execute(f"""
        CREATE TABLE activities AS SELECT id, type,
          try_cast(replace(created_at, 'Z', '') AS TIMESTAMP) AS ts,
          actor.type AS actor_type, lower(trim(actor.email_address)) AS actor_email,
          actor.user_id AS actor_user_id,
          lower(trim("{TARGET_FIELD}")) AS target_email, coalesce(deleted_user_id, user_id) AS
target_user_id
        FROM raw""")
    con.execute(f"""CREATE TABLE leavers AS SELECT lower(trim("{LEAVER_EMAIL_COL}")) AS email,
          trim(coalesce("{LEAVER_DATE_COL}", '')) AS last_day_text, {strict_date(LEAVER_DATE_COL)}
AS last_day
        FROM read_csv(?, header=true, all_varchar=true)""", [lvr])
    con.execute(f"""CREATE TABLE reviewers AS SELECT lower(trim("{REVIEWER_EMAIL_COL}")) AS email
          FROM read_csv(?, header=true, all_varchar=true)""", [rost])
    con.execute(f"""CREATE TABLE reviews_raw AS SELECT
          {strict_date(REVIEW_START_COL)} AS period_start, {strict_date(REVIEW_END_COL)} AS
period_end,
          lower(trim("{REVIEWER_COL}")) AS reviewer, {strict_date(SIGNED_COL)} AS signed_on,
          trim(coalesce("{SIGNED_COL}", '')) AS signed_text
          FROM read_csv(?, header=true, all_varchar=true)""", [rvw])
    p0, p1 = PERIOD_START, PERIOD_END
    en = f"DATE '{enabled}'" if enabled else "NULL::DATE"
    rt = ", ".join(f"'{t}'" for t in REMOVAL_TYPES)
    # who each removal record names: the email field, else the id mapped through actor.user_id,
else
    # (org_user_left) the person who removed themselves
    con.execute(f"""
        CREATE TABLE removals AS
        WITH ids AS (SELECT actor_user_id, arg_max(actor_email, ts) AS email FROM activities
                     WHERE actor_type = 'user_actor' AND actor_user_id IS NOT NULL AND actor_email
IS NOT NULL GROUP BY 1)
        SELECT a.id, a.ts, a.type, coalesce(a.target_email, i.email,
               CASE WHEN a.type = 'org_user_left' THEN a.actor_email END) AS email
        FROM activities a LEFT JOIN ids i ON i.actor_user_id = a.target_user_id WHERE a.type IN
({rt})""")
    # a review row is usable only if both ends of its period read as dates; unusable rows are
counted below
    con.execute("CREATE TABLE reviews AS SELECT * FROM reviews_raw WHERE period_start IS NOT NULL
AND period_end IS NOT NULL")

    # ---- input checks: each says which planted fault trips it
    total = con.execute("SELECT count(*) FROM activities").fetchone()[0]
    dup = con.execute("SELECT count(*) FROM (SELECT id FROM activities GROUP BY id HAVING count(*)
> 1)").fetchone()[0]
```

```
    no_time = con.execute("SELECT count(*) FROM activities WHERE ts IS NULL OR id IS
NULL").fetchone()[0]
    outside = con.execute(f"SELECT count(*) FROM activities WHERE ts::DATE NOT BETWEEN '{p0}' AND
'{p1}'").fetchone()[0]
    days = con.execute(f"""
        SELECT d::DATE AS day, count(a.id) AS records,
               CASE WHEN count(a.id) > 0 THEN 'recorded'
                    WHEN d::DATE < {en} THEN 'BEFORE ENABLED' ELSE 'NO RECORDS' END AS status
        FROM generate_series(DATE '{p0}', DATE '{p1}', INTERVAL 1 DAY) t(d)
        LEFT JOIN activities a ON a.ts::DATE = d::DATE GROUP BY 1 ORDER BY 1""").fetchall()
    off = [r[0] for r in days if r[2] == "NO RECORDS"]
    pre = [r[0] for r in days if r[2] == "BEFORE ENABLED"]
    bad_review = con.execute("SELECT count(*) FROM reviews_raw WHERE period_start IS NULL OR
period_end IS NULL").fetchone()[0]
    bad_signed = con.execute("SELECT count(*) FROM reviews_raw WHERE signed_text <> '' AND
signed_on IS NULL").fetchone()[0]
    ros_bad = con.execute("SELECT count(*) FROM reviewers WHERE email IS NULL OR email =
''").fetchone()[0]
    hr_bad = con.execute("SELECT count(*) FROM leavers WHERE email IS NULL OR email = '' OR
last_day_text = ''").fetchone()[0]
    hr_date = con.execute("SELECT count(*) FROM leavers WHERE last_day_text <> '' AND last_day IS
NULL").fetchone()[0]
    checks = [
        ("duplicate activity ids", dup, "a duplicated record"),
        ("records with no id or no readable created_at", no_time, "an unreadable timestamp"),
        ("days on or after enablement with no records", len(off), "five days with no records"),
        ("days in the period before the Compliance API was enabled", len(pre), "the Compliance API
turned on late"),
        ("review log rows with no readable period start or end", bad_review, "a review row with no
end date"),
        ("review log rows with a signature date that is not YYYY-MM-DD", bad_signed, "a signature
date written 09/24/2026"),
        ("reviewer roster rows with no email", ros_bad, "a NULL in the roster"),
        ("HR leaver rows with no email or no last day", hr_bad, "a NULL in the HR file"),
        ("HR leaver rows whose last day is not written as YYYY-MM-DD", hr_date, "a last day written
8/5/26"),
    ]
    print(f"\n== Input checks   (records read: {total}; outside the period, left out: {outside})")
    ok = True
    for label, n, fault in checks:
        flag = "ok  " if n == 0 else "FAIL"
        ok &= n == 0
        print(f"[{flag}] {label}: {n}" + ("" if n == 0 else f"   (trips on: {fault})"))
    for label, ds in [("days with no records", off), ("days before enablement", pre)]:
        if ds:
            print(f"        {label}: " + ", ".join(f"{a}..{b}" if a != b else str(a) for a, b, _ in
ranges(ds)))
    seen = {r[0] for r in con.execute("SELECT DISTINCT type FROM activities").fetchall()}
    print("[info] removal types never seen in this feed: " + (", ".join(t for t in REMOVAL_TYPES if
t not in seen) or "none"))
    unnamed = con.execute("SELECT count(*) FROM removals WHERE email IS NULL").fetchone()[0]
    print(f"[info] removal records that name no email the script can match: {unnamed}")

    # ---- 164.312(b): was the feed recording on every day of the period?
    r_log = population(con, "audit_controls_daily", "164.312(b)",
                       "one row per day; records counted by created_at date; enabled on " +
(enabled or "not known"),
                       f"""SELECT d::DATE AS day, count(a.id) AS records,
```

```
                                    CASE WHEN count(a.id) > 0 THEN 'recorded'
                                         WHEN d::DATE < {en} THEN 'BEFORE ENABLED' ELSE 'NO RECORDS'
END AS status
                            FROM generate_series(DATE '{p0}', DATE '{p1}', INTERVAL 1 DAY) t(d)
                            LEFT JOIN activities a ON a.ts::DATE = d::DATE GROUP BY 1 ORDER BY 1""",
                        ["day", "records", "status"], ok)
    n_rec = sum(1 for r in r_log if r[2] == "recorded")
    print(f"  days recorded: {n_rec} of {len(r_log)}")
    for label, st in [("NO RECORDS", "NO RECORDS"), ("BEFORE ENABLED", "BEFORE ENABLED")]:
        for a, b, n in ranges([r[0] for r in r_log if r[2] == st]):
            print(f"  {label}: {a} to {b} ({n} day(s))")

    # ---- 164.308(a)(1)(ii)(D): every review, and who signed it when
    sql = f"""
        SELECT v.period_start, v.period_end, coalesce(v.reviewer, '-') AS reviewer, v.signed_on,
               date_diff('day', v.period_end, v.signed_on) AS days_to_sign,
               CASE WHEN v.signed_on IS NULL AND v.signed_text <> '' THEN 'SIGNATURE DATE
UNREADABLE'
                    WHEN v.signed_on IS NULL THEN 'NOT SIGNED'
                    WHEN v.signed_on < v.period_end THEN 'SIGNED BEFORE PERIOD ENDED'
                    WHEN v.reviewer IS NULL OR v.reviewer = '' THEN 'NO REVIEWER'
                    WHEN v.reviewer NOT IN (SELECT email FROM reviewers WHERE email IS NOT NULL)
THEN 'NOT ON ROSTER'
                    WHEN date_diff('day', v.period_end, v.signed_on) > {SIGN_WITHIN_DAYS} THEN
'LATE'
                    ELSE 'on time' END AS status
        FROM reviews v WHERE v.period_start <= DATE '{p1}' AND v.period_end >= DATE '{p0}'
        ORDER BY v.period_start, v.period_end, v.reviewer"""
    r_rev = population(con, "activity_reviews", "164.308(a)(1)(ii)(D)",
                       f"review log rows overlapping the period; signed within {SIGN_WITHIN_DAYS}
day(s) of the period's last day; "
                       "reviewer on the roster", sql,
                       ["period_start", "period_end", "reviewer", "signed_on", "days_to_sign",
"status"], ok)
    for r in r_rev:
        print("  " + " | ".join("-" if v is None else str(v) for v in r))
    sql = f"""
        WITH d AS (SELECT d::DATE AS day FROM generate_series(DATE '{p0}', DATE '{p1}', INTERVAL 1
DAY) t(d)
                   WHERE NOT EXISTS (SELECT 1 FROM reviews v WHERE v.period_start <= d::DATE AND
v.period_end >= d::DATE)),
             g AS (SELECT day, day - (row_number() OVER (ORDER BY day))::INTEGER AS grp FROM d)
        SELECT min(day) AS first_day, max(day) AS last_day, count(*) AS days FROM g GROUP BY grp
ORDER BY 1"""
    r_gap = population(con, "days_without_review", "164.308(a)(1)(ii)(D)",
                       "days of the period that no readable review row covers", sql, ["first_day",
"last_day", "days"], ok)
    for r in r_gap:
        print("  " + " | ".join(str(v) for v in r))

    # ---- 164.308(a)(3)(ii)(C): leavers. Each removal record is credited to ONE departure of that
person: the
    # latest HR last day on or before it, or, if it came before every last day, the earliest last
day it precedes
    # by at most REMOVAL_WINDOW_DAYS. So a removal from an earlier stint never counts for a later
departure.
    con.execute(f"""
        CREATE TABLE had_account AS
```

```
        SELECT actor_email AS email FROM activities
        WHERE actor_type = 'user_actor' AND actor_email IS NOT NULL AND ts::DATE <= DATE '{p1}'
        UNION SELECT email FROM removals WHERE email IS NOT NULL""")
    in_period = f"l.last_day BETWEEN DATE '{p0}' AND DATE '{p1}' AND l.email IS NOT NULL"
    sql = f"""
        WITH credited AS (
              SELECT r.email, r.ts, coalesce(
                (SELECT max(l.last_day) FROM leavers l WHERE l.email = r.email AND l.last_day <=
r.ts::DATE),
                (SELECT min(l.last_day) FROM leavers l WHERE l.email = r.email
                  AND l.last_day - INTERVAL {REMOVAL_WINDOW_DAYS} DAY <= r.ts::DATE)) AS last_day
              FROM removals r WHERE r.email IS NOT NULL),
             rem AS (SELECT email, last_day, min(ts) AS removed_at FROM credited
                     WHERE last_day IS NOT NULL GROUP BY 1, 2),
             nxt AS (SELECT email, last_day,
                            lead(last_day) OVER (PARTITION BY email ORDER BY last_day) AS
next_last_day
                     FROM leavers WHERE email IS NOT NULL),
             -- after the grace period and before the removal; and after the removal. When the same
person
             -- has a LATER departure in HR, use after the removal belongs to that later stint, not
this row.
             after AS (SELECT l.email, l.last_day,
                         count(a.id) FILTER (WHERE a.ts::DATE > l.last_day + INTERVAL {GRACE_DAYS}
DAY
                                                 AND (r.removed_at IS NULL OR a.ts < r.removed_at))
AS n_grace,
                         count(a.id) FILTER (WHERE r.removed_at IS NOT NULL AND a.ts > r.removed_at
                                                 AND n.next_last_day IS NULL) AS n_removed
                       FROM leavers l JOIN nxt n ON n.email = l.email AND n.last_day = l.last_day
                       LEFT JOIN rem r ON r.email = l.email AND r.last_day = l.last_day
                       LEFT JOIN activities a ON a.actor_email = l.email AND a.actor_type =
'user_actor'
                       GROUP BY 1, 2)
        SELECT l.email, l.last_day, r.removed_at::DATE AS first_removal_record,
               date_diff('day', l.last_day, r.removed_at::DATE) AS days_to_removal,
               af.n_grace AS actions_after_grace, af.n_removed AS actions_after_removal,
               CASE WHEN r.removed_at IS NULL AND l.email IN (SELECT email FROM had_account) THEN
'NO REMOVAL RECORD'
                    WHEN r.removed_at IS NULL THEN 'CHECK: NO CLAUDE ACTIVITY SEEN'
                    WHEN af.n_removed > 0 THEN 'ACTIVE AFTER REMOVAL'
                    WHEN date_diff('day', l.last_day, r.removed_at::DATE) > {GRACE_DAYS} THEN
'REMOVED LATE'
                    ELSE 'removed in time' END AS status
        FROM leavers l
        LEFT JOIN rem r ON r.email = l.email AND r.last_day = l.last_day
        LEFT JOIN after af ON af.email = l.email AND af.last_day = l.last_day
        WHERE {in_period}
        ORDER BY l.last_day, l.email"""
    r_lv = population(con, "leavers", "164.308(a)(3)(ii)(C)",
                      f"HR leavers with last day in period; removal types {REMOVAL_TYPES}, each
credited to one departure; "
                      f"grace {GRACE_DAYS} day(s)", sql,
                      ["email", "last_day", "first_removal_record", "days_to_removal",
"actions_after_grace",
                       "actions_after_removal", "status"], ok)
    for r in r_lv:
        print("  " + " | ".join("-" if v is None else str(v) for v in r))
```

```
    follow = (len(off) + len(pre) + sum(1 for r in r_rev if r[5] != "on time") + len(r_gap)
              + sum(1 for r in r_lv if r[6] != "removed in time" or r[4] or r[5]))
    print("\n== Result")
    print(f"input checks : {'all passed' if ok else 'FAILED, the lists above are not complete'}")
    print(f"follow-up    : {follow} item(s) need an explanation "
          f"(days with no records or before enablement, reviews not on time, days with no review,
leaver rows)")
    verdict = "NOTHING TO FOLLOW UP" if ok and follow == 0 else "EXCEPTIONS FOUND"
    print(f"RESULT: {verdict}")
    return 1 if strict and verdict != "NOTHING TO FOLLOW UP" else 0


if __name__ == "__main__":
    sys.exit(main())
```

The runs

Run 1, with faults planted: `python3 hipaa_claude_review_evidence.py --plant`

```
input     : GENERATED records (not a real organization), seed 222, plant=True
enabled on: 2026-07-08

== Input checks   (records read: 504; outside the period, left out: 0)
[FAIL] duplicate activity ids: 1   (trips on: a duplicated record)
[FAIL] records with no id or no readable created_at: 1   (trips on: an unreadable timestamp)
[FAIL] days on or after enablement with no records: 5   (trips on: five days with no records)
[FAIL] days in the period before the Compliance API was enabled: 2   (trips on: the Compliance API
turned on late)
[FAIL] review log rows with no readable period start or end: 1   (trips on: a review row with no
end date)
[FAIL] review log rows with a signature date that is not YYYY-MM-DD: 1   (trips on: a signature
date written 09/24/2026)
[FAIL] reviewer roster rows with no email: 1   (trips on: a NULL in the roster)
[FAIL] HR leaver rows with no email or no last day: 1   (trips on: a NULL in the HR file)
[FAIL] HR leaver rows whose last day is not written as YYYY-MM-DD: 1   (trips on: a last day
written 8/5/26)
        days with no records: 2026-08-10..2026-08-14
        days before enablement: 2026-07-06..2026-07-07
[info] removal types never seen in this feed: scim_user_deleted, org_user_left
[info] removal records that name no email the script can match: 0

== 164.312(b): audit_controls_daily   ** INCOMPLETE: see input checks **
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : one row per day; records counted by created_at date; enabled on 2026-07-08
run at    : 2026-10-31 15:51:36 UTC
rows      : 84
sha256    : 34b1b8a863fd97d107f200b9cdb3072d2e7557c5b02623b808a10c36f53f71b0
  days recorded: 77 of 84
  NO RECORDS: 2026-08-10 to 2026-08-14 (5 day(s))
  BEFORE ENABLED: 2026-07-06 to 2026-07-07 (2 day(s))

== 164.308(a)(1)(ii)(D): activity_reviews   ** INCOMPLETE: see input checks **
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : review log rows overlapping the period; signed within 14 day(s) of the period's last
```

```
day; reviewer on the roster
run at    : 2026-10-31 15:51:36 UTC
rows      : 10
sha256    : 3346ac082bbbed865ffdb52c2b2eef744b275fb80abd04b6584aa754eb8695a1
  2026-07-06 | 2026-07-12 | privacy.officer@example.test | 2026-07-15 | 3 | on time
  2026-07-13 | 2026-07-19 | privacy.officer@example.test | - | - | SIGNATURE DATE UNREADABLE
  2026-07-20 | 2026-07-26 | privacy.officer@example.test | 2026-07-29 | 3 | on time
  2026-08-03 | 2026-08-09 | security.lead@example.test | 2026-08-10 | 1 | on time
  2026-08-10 | 2026-08-16 | privacy.officer@example.test | 2026-08-19 | 3 | on time
  2026-08-17 | 2026-08-23 | security.lead@example.test | 2026-09-25 | 33 | LATE
  2026-08-24 | 2026-08-30 | security.lead@example.test | 2026-09-02 | 3 | on time
  2026-08-31 | 2026-09-06 | u11@example.test | 2026-09-08 | 2 | NOT ON ROSTER
  2026-09-14 | 2026-09-20 | security.lead@example.test | - | - | NOT SIGNED
  2026-09-21 | 2026-09-27 | privacy.officer@example.test | 2026-09-24 | -3 | SIGNED BEFORE PERIOD
ENDED

== 164.308(a)(1)(ii)(D): days_without_review   ** INCOMPLETE: see input checks **
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : days of the period that no readable review row covers
run at    : 2026-10-31 15:51:36 UTC
rows      : 2
sha256    : 7f9d75842e8da1d914017c022e98e8cddd2065c3bba91cb5e93e2b441fdc09c3
  2026-07-27 | 2026-08-02 | 7
  2026-09-07 | 2026-09-13 | 7

== 164.308(a)(3)(ii)(C): leavers   ** INCOMPLETE: see input checks **
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : HR leavers with last day in period; removal types ['org_user_deleted',
'scim_user_deleted', 'org_user_left'], each credited to one departure; grace 1 day(s)
run at    : 2026-10-31 15:51:36 UTC
rows      : 6
sha256    : 41a3adb2e3c845161632c77e961990a0248a6515bd8ac4ad3b0d1ddbf2eefa98
  u30@example.test | 2026-07-10 | 2026-07-11 | 1 | 0 | 1 | ACTIVE AFTER REMOVAL
  u31@example.test | 2026-07-21 | 2026-07-22 | 1 | 0 | 0 | removed in time
  u32@example.test | 2026-08-01 | 2026-08-01 | 0 | 0 | 0 | removed in time
  u33@example.test | 2026-08-12 | - | - | 10 | 0 | NO REMOVAL RECORD
  u34@example.test | 2026-08-23 | 2026-09-01 | 9 | 0 | 0 | REMOVED LATE
  u35@example.test | 2026-09-03 | 2026-09-03 | 0 | 0 | 0 | removed in time

== Result
input checks : FAILED, the lists above are not complete
follow-up    : 17 item(s) need an explanation (days with no records or before enablement, reviews
not on time, days with no review, leaver rows)
RESULT: EXCEPTIONS FOUND
```

Run 2, clean records: `python3 hipaa_claude_review_evidence.py`

```
input     : GENERATED records (not a real organization), seed 222, plant=False
enabled on: 2026-07-06

== Input checks   (records read: 534; outside the period, left out: 0)
[ok  ] duplicate activity ids: 0
[ok  ] records with no id or no readable created_at: 0
[ok  ] days on or after enablement with no records: 0
[ok  ] days in the period before the Compliance API was enabled: 0
[ok  ] review log rows with no readable period start or end: 0
[ok  ] review log rows with a signature date that is not YYYY-MM-DD: 0
```

```
[ok  ] reviewer roster rows with no email: 0
[ok  ] HR leaver rows with no email or no last day: 0
[ok  ] HR leaver rows whose last day is not written as YYYY-MM-DD: 0
[info] removal types never seen in this feed: scim_user_deleted, org_user_left
[info] removal records that name no email the script can match: 0

== 164.312(b): audit_controls_daily
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : one row per day; records counted by created_at date; enabled on 2026-07-06
run at    : 2026-10-31 15:51:37 UTC
rows      : 84
sha256    : b8badaea15e885832a711ab9aa1122dc976b1df2aba52f9a3c6c9a17cb5f9534
  days recorded: 84 of 84

== 164.308(a)(1)(ii)(D): activity_reviews
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : review log rows overlapping the period; signed within 14 day(s) of the period's last
day; reviewer on the roster
run at    : 2026-10-31 15:51:37 UTC
rows      : 12
sha256    : c979dc4fe60649cf382b64079e1580fc0fddb333779ef5a79f8d9f17014863e6
  2026-07-06 | 2026-07-12 | privacy.officer@example.test | 2026-07-14 | 2 | on time
  2026-07-13 | 2026-07-19 | privacy.officer@example.test | 2026-07-20 | 1 | on time
  2026-07-20 | 2026-07-26 | privacy.officer@example.test | 2026-07-30 | 4 | on time
  2026-07-27 | 2026-08-02 | security.lead@example.test | 2026-08-03 | 1 | on time
  2026-08-03 | 2026-08-09 | privacy.officer@example.test | 2026-08-10 | 1 | on time
  2026-08-10 | 2026-08-16 | security.lead@example.test | 2026-08-22 | 6 | on time
  2026-08-17 | 2026-08-23 | security.lead@example.test | 2026-08-26 | 3 | on time
  2026-08-24 | 2026-08-30 | privacy.officer@example.test | 2026-09-01 | 2 | on time
  2026-08-31 | 2026-09-06 | security.lead@example.test | 2026-09-10 | 4 | on time
  2026-09-07 | 2026-09-13 | security.lead@example.test | 2026-09-19 | 6 | on time
  2026-09-14 | 2026-09-20 | security.lead@example.test | 2026-09-23 | 3 | on time
  2026-09-21 | 2026-09-27 | security.lead@example.test | 2026-09-29 | 2 | on time

== 164.308(a)(1)(ii)(D): days_without_review
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : days of the period that no readable review row covers
run at    : 2026-10-31 15:51:37 UTC
rows      : 0
sha256    : e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

== 164.308(a)(3)(ii)(C): leavers
period    : 2026-07-06 to 2026-09-27 (UTC, inclusive)
filters   : HR leavers with last day in period; removal types ['org_user_deleted',
'scim_user_deleted', 'org_user_left'], each credited to one departure; grace 1 day(s)
run at    : 2026-10-31 15:51:37 UTC
rows      : 6
sha256    : cea08b0c74c6922c14cb3ad10fea4943175d430ce38654138a1a304aeff19144
  u30@example.test | 2026-07-10 | 2026-07-11 | 1 | 0 | 0 | removed in time
  u31@example.test | 2026-07-21 | 2026-07-21 | 0 | 0 | 0 | removed in time
  u32@example.test | 2026-08-01 | 2026-08-01 | 0 | 0 | 0 | removed in time
  u33@example.test | 2026-08-12 | 2026-08-12 | 0 | 0 | 0 | removed in time
  u34@example.test | 2026-08-23 | 2026-08-23 | 0 | 0 | 0 | removed in time
  u35@example.test | 2026-09-03 | 2026-09-03 | 0 | 0 | 0 | removed in time

== Result
input checks : all passed
follow-up    : 0 item(s) need an explanation (days with no records or before enablement, reviews
```

```
not on time, days with no review, leaver rows)
RESULT: NOTHING TO FOLLOW UP
```

How to read the results

- **INCOMPLETE in a list heading** means an input check failed. Fix the input first; do not hand the list over as it is.
- **NO RECORDS** is a day with no feed records on or after the day you turned the Compliance API on. Find out whether recording was off or the day was quiet.
- **BEFORE ENABLED** is a day before the Compliance API was on. The feed cannot evidence it. Say so.
- **SIGNATURE DATE UNREADABLE, LATE, NOT SIGNED, SIGNED BEFORE PERIOD ENDED, NOT ON ROSTER, NO REVIEWER** are review rows that need an explanation or a correction in your own log.
- **Days without review** are days no readable row covers. A row with an unreadable date also leaves its days here.
- **NO REMOVAL RECORD, REMOVED LATE, ACTIVE AFTER REMOVAL, CHECK: NO CLAUDE ACTIVITY SEEN** are leaver rows that need an explanation. Clear each CHECK row against your seat list.
- **RESULT: NOTHING TO FOLLOW UP** says only that the input passed its checks and no row needed an explanation. It does not say a requirement is satisfied.

What the Security Rule text says

Three provisions in 45 CFR part 164 are the ones a review of Claude activity touches. The text below is quoted from 45 CFR part 164, subpart C, eCFR text as of 2026-10-15 (read 2026-10-31).

- **164.312(b), Audit controls (a standard):** “Implement hardware, software, and/or procedural mechanisms that record and examine activity in information systems that contain or use electronic protected health information.”
- **164.308(a)(1)(ii)(D), Information system activity review (Required):** “Implement procedures to regularly review records of information system activity, such as audit logs, access reports, and security incident tracking reports.”
- **164.308(a)(3)(ii)(C), Termination procedures (Addressable):** “Implement procedures for terminating access to electronic protected health information when the employment of, or other arrangement with, a workforce member ends or as required by determinations made as specified in paragraph (a)(3)(ii)(B) of this section.”

The text we quote sets no review frequency and no deadline for signing a review. It does say "regularly", so choose a frequency you can justify, write it into your own procedure, and keep your records matching that procedure. Section 164.316(b)(2)(i) separately asks that documentation the rule requires be kept for 6 years from the date it was created or last in effect, whichever is later. That period covers required documentation, not necessarily raw logs; ask your counsel what applies to the logs. OCR's audit protocol for 164.308(a)(1)(ii)(D) (archived copy, read 2026-10-31) asks for documentation of the records that were reviewed, and checks whether they were reviewed in a timely manner by appropriate personnel. For 164.312(b) it checks whether activity in systems with ePHI is being recorded and examined. The lists below follow those questions. The protocol is an audit tool, not a checklist a review passes.

This guide is about Claude Enterprise only. For the same Security Rule rated against the logs of a data platform, see HIPAA Security Rule evidence from data platform logs.

What this covers, and what it does not

Anthropic offers a HIPAA-ready version of Claude for organizations on Enterprise plans, which includes a Business Associate Agreement (Anthropic: HIPAA-ready Claude Enterprise offering (read 2026-10-31)). That agreement is between you and Anthropic. Data Meaning is not a party to it. Whether your use of Claude with patient data is covered is for your privacy officer and counsel. What your Business Associate Agreement covers is a separate question from what the Compliance API records.

The Compliance API covers Claude Enterprise. For Claude Console and API-key workloads, the Activity Feed records administrative and resource events but no prompt or response text. Claude Code sessions authenticated with a Console API key, run through Amazon Bedrock, Google Cloud or Microsoft Foundry, or run as Claude Code cloud sessions are not captured (Anthropic: Compliance API FAQ (read 2026-10-31)). Recording starts when the Compliance API is first turned on and is not backfilled (Anthropic: Query the Activity Feed (read 2026-10-31)). Anything outside that coverage needs its own evidence.

What you will have at the end

A Python script, hipaa_claude_review_evidence.py, that reads your Claude Activity Feed pull, your activity review log, your roster of people allowed to sign a review and an HR leaver list. It runs nine input checks, then builds four lists and prints each with its period, its filters, the run time, its row count and a SHA-256 of its rows. A reader with the same input and the same script gets the same row count and the same fingerprint. It also writes each list as a CSV. The full detail is in a PDF: Download the full guide (PDF).

- **164.312(b):** one row for each day of the period, marked `recorded`, `NO RECORDS` or `BEFORE ENABLED`.

- **164.308(a)(1)(ii)(D):** every review in your log with its reviewer, the date signed and a status, and every stretch of days that no review covers.
- **164.308(a)(3)(ii)(C):** every leaver, the days to the first removal record, and any use of the account after it.

The records we ran it on are **generated**: a seeded stand-in, not a real organization's data and not a measurement of anything. No real organization's feed is public. The field names come from Anthropic's published Activity object, so the script is built to read a real pull. We have not run it on one.

This page produces evidence. It does not say a requirement is satisfied or that anyone will pass a review. We name specifications by id and quote only the public regulation text.

Checklist

Done	Item	Where
☐	The Compliance API is on, and you know the day you first turned it on	Set COMPLIANCE_API_ENABLED_ON
☐	The Activity Feed pull covers the whole period, with its cursor, record count, run time and final request-id kept	Step 1
☐	Duplicate ids are removed before the run	Step 3
☐	All nine input checks pass, or you can explain each failure	Step 3
☐	Every day is recorded; each NO RECORDS stretch has an explanation	Step 4
☐	Each BEFORE ENABLED day is stated as not evidenced by the feed	Step 4
☐	Your procedure names the review frequency and the signing limit, and SIGN_WITHIN_DAYS matches it	Step 5
☐	Every review is signed within your limit by someone on the roster, or has an explanation	Step 5
☐	No stretch of days is without a review, or each one has an explanation	Step 6
☐	Every leaver row is removed in time, or has an explanation	Step 7
☐	The fingerprints and CSV files are saved with the run's parameters	Step 7
☐	Your own copy of the pull, the review log and the lists is kept as long as your procedure says	Limits

Run it on your own data

To run this on your own organization, save your Activity Feed pull as one JSON object per line, and save your review log, your roster of people allowed to sign a review and your HR leaver list as CSV files with a header row, with every date written YYYY-MM-DD. At the top of the script, set ACTIVITIES_FILE, REVIEWS_FILE, REVIEWERS_FILE and LEAVERS_FILE to those file paths, PERIOD_START and PERIOD_END to the period you are evidencing, COMPLIANCE_API_ENABLED_ON to the day you first turned the Compliance API on, REVIEW_START_COL, REVIEW_END_COL, REVIEWER_COL, SIGNED_COL, REVIEWER_EMAIL_COL, LEAVER_EMAIL_COL and LEAVER_DATE_COL to your CSV column names, and SIGN_WITHIN_DAYS to the number of days your procedure allows for signing a review. If your removal records hold the removed user email in a field other than deleted_user_email, set TARGET_FIELD to the name of that field. Change nothing else.

We tested those steps on a separate set of files: different file names in a folder whose name has an apostrophe, different column names, mixed-case emails, a differently named removal field, a NULL in the roster and in the HR file, and a signing limit changed from 14 to 10 days. The faults planted were a duplicated record, three days with no records, two days before enablement, a week with no review, a review signed after the limit, a reviewer not on the roster, an unsigned review, a review row with no end date, a signature date not written YYYY-MM-DD, a last day not written YYYY-MM-DD, and three leaver faults. The script caught every one we planted and matched the same person across different email capitalization.

Download hipaa_claude_review_evidence.py
Download the full guide (PDF)

Limits

- **This is not a client result.** The records are generated. The run shows the script finds planted faults. It says nothing about any real organization or about how often such faults happen.
- **We did not call the Compliance API.** Real pulls can differ from the documentation: other field names for the removed user, other activity types for removals, and records for service accounts.
- **The lists are our reading** of the rule text, not an official list from HHS. The review frequency and the signing limit are not set by the text we quote; they are your procedure's numbers.
- **A signature is not a review.** The script checks that a row exists, who signed it and when. It cannot tell whether the reviewer looked at anything, or what they decided about what they saw.
- **Coverage:** only what the Compliance API records, from the day it was first turned on. For Claude Console and API-key workloads, the Activity Feed records administrative and resource

events but no prompt or response text. Claude Code sessions authenticated with a Console API key, run through Amazon Bedrock, Google Cloud or Microsoft Foundry, or run as Claude Code cloud sessions are not captured (Anthropic: Compliance API FAQ (read 2026-10-31)).

- **Content is kept for a different time than the feed.** Activity Feed records are retained for 6 years, but in organizations with HIPAA readiness enabled, local session data for Cowork and Claude Code is stored for 30 days (Anthropic: Compliance API FAQ (read 2026-10-31)). Anthropic advises exporting what you must keep longer (Anthropic: Plan content retention (read 2026-10-31)). The script reads files and keeps nothing itself, so keep your own copy of the pull, the review log and the lists. The pull holds workforce emails and other identifying details, so store it under the access controls you use for other security records. The script runs on your machine and sends nothing to Data Meaning.
- **The empty-day check can fire on a quiet day,** and cannot see a day that has some records but not all.
- **Each removal is credited to one departure,** and a removal more than 30 days before every last day shows as `NO REMOVAL RECORD` and needs a look. The SOC 2 guide explains the cases.
- **A removal record is not proof of logout.** The script counts later actions by the same account as a second signal, but a person can also use an API key that is not tied to their name.

Method

- **Input:** generated Activity Feed records, written by the script as JSON lines in the Activity object's shape, with a fixed seed, plus a generated review log, roster and HR list. Fields used: `id`, `created_at`, `type`, `actor.type`, `actor.email_address`, `actor.user_id` (Anthropic: Activity object (read 2026-10-31)), plus `deleted_user_email`, `deleted_user_id` and `user_id` on removal records (Anthropic: API reference (read 2026-10-31)).
- **Period:** 2026-07-06 to 2026-09-27, UTC, inclusive: twelve Monday-to-Sunday weeks. Days are counted by the `created_at` date in UTC. Leaver removals are looked for in the whole file, because a removal can come after the period ends.
- **Lists:** the logging list starts from the calendar, the review lists from your review log, and the leaver list from the HR list. Rows are never silently dropped: a failed input check marks the lists INCOMPLETE, and a review row that cannot be read is counted.
- **Settings you can disagree with** are written at the top of the script: the signing limit (14 days), the grace period for removals (1 day) and the removal window (30 days).

Tested on

Tested on

Platform
: DuckDB 1.5.6, Python 3.12, run from Python with the `duckdb` module

Edition
: Open-source DuckDB, in-memory

Date
: 2026-10-31

Dataset
: Generated records, seed 222 (not a real organization). Field names from Anthropic's Activity object (read 2026-10-31)

Code
: hipaa_claude_review_evidence.py

When this is not enough

- **You need every period, not one run.** The script is a point-in-time pull. Keep each pull's parameters and the lists as they were produced.
- **Claude is used outside the Compliance API's coverage.** Those uses need evidence from the platform they run on.
- **The requirement is more than a list.** Your risk analysis, your written procedures, your business associate agreement and what you did about each finding sit in your own records, not in the feed.
- **Legal advice is not here.** Whether a list is enough is for your privacy officer and counsel.

Sources

- 45 CFR part 164, subpart C, eCFR text as of 2026-10-15 (read 2026-10-31)
- Anthropic: HIPAA-ready Claude Enterprise offering (read 2026-10-31)
- Anthropic: Query the Activity Feed (read 2026-10-31)
- Anthropic: Design your compliance integration (read 2026-10-31)
- Anthropic: Compliance API FAQ (read 2026-10-31)
- Anthropic: Query compliance activities, API reference (read 2026-10-31)
- HHS OCR audit protocol (archived 2026-01-08) (read 2026-10-31)

- DuckDB: loading JSON (read 2026-10-31)

Control Tower for Claude

Control Tower for Claude runs in your own Azure subscription. See how it fits your Claude Enterprise setup.

https://www.datameaning.com/solutions/claude-control-tower.html