DATA CONTROL TOWER · GUIDE

SOC 2 User Access Evidence for Snowflake, Databricks, Redshift

Three lists a SOC 2 user access review can draw on, and a check for missing users: a DuckDB script, run on generated records.

Published 2026-11-01 · Online version and downloads: www.datameaning.com/guides/soc-2-user-access-evidence-for-snowflake-databricks-redshift/

In our reading, SOC 2 criteria CC6.1, CC6.2 and CC6.3 are about three things you can list: the user accounts, the administrators, and the successful sign-ins by method. Your auditor decides what to sample, so ask for their request. This guide shows a free script that builds all three from Snowflake, Databricks and Redshift exports, on demo records.

This guide goes with the page https://www.datameaning.com/guides/soc-2-user-access-evidence-for-snowflake-databricks-redshift/. It holds the details the page keeps short. Everything here comes from the script and the runs on that page; the records are generated, not a company's data, and the script has not been run on a real account.

What the criteria cover, in our words

The AICPA's Trust Services Criteria are licensed text, so this page names criteria by id and describes them in our own words; it quotes none of them (AICPA: 2017 Trust Services Criteria (With Revised Points of Focus, 2022) (read 2026-11-01)). CC6.1 is about logical access security, including identifying and authenticating users. Multifactor authentication appears in one of its points of focus, as an example of stronger authentication where an organization's own risk strategy calls for it, so this page never says SOC 2 requires it: your own policy decides, which is why the sign-in policy in the script is a setting with no default. CC6.2 is about registering and authorizing users before they are given credentials, and removing credentials when access is no longer authorized; its points of focus include reviewing credentials for validity, including system or service accounts that should not be there. CC6.3 is about authorizing, changing and removing access according to people's roles and duties, or how the system is built, with regard to least privilege and segregation of duties.

Points of focus describe characteristics, not requirements. We do not know what any particular auditor will sample, and this page does not say. What follows is our reading: those three criteria are about who has an account, who can administer, and how people authenticate, so each of those is a list you can build from the platforms' own records, and the first list is only worth showing if you can also show it is not missing anyone. Ask your auditor for their sample request, then check it against the lists below. Approvals before a grant and the timing of a leaver's removal are in our SOX access evidence guide; this page does not repeat them.

What you will need

- Access to the platforms' own records. On Redshift, SYS_CONNECTION_LOG is "visible only to superusers", and SVV_USER_INFO shows every row only to a superuser, so export both as a superuser (Amazon Redshift: SYS_CONNECTION_LOG (read 2026-11-01); Amazon Redshift: SVV_USER_INFO (read 2026-11-01)). An empty connection-log export fails an input check. On Snowflake, the ACCOUNT_USAGE views USERS, LOGIN_HISTORY and GRANTS_TO_USERS can each lag by up to 120 minutes, so take the export some hours after the last event you need (Snowflake: USERS (read 2026-11-01); Snowflake: LOGIN_HISTORY (read 2026-11-01); Snowflake: GRANTS_TO_USERS (read 2026-11-01)).
- Your HR file, a login map, a list of service accounts with an owner each, a list of approved administrators, and your sign-in policy.
- Python 3 and the `duckdb` module (we ran it with duckdb 1.5.6 and, for the clean run, 2.0.0).

Step 1. Export each platform's records into one folder

The script reads these files from `EXPORT_DIR`. Column names are the platforms' own; the script lower-cases the header, so `CREATED_ON` and `created_on` both work, and it stops and says which column is missing if one is not there. Databricks' list-users API returns `userName`, `active` and `groups` for each user, at most 100 per page, so page through with `startIndex` until you have them all (Databricks: list users (SCIM) (read 2026-11-01)); flatten the groups to their names. Service principals are a separate SCIM resource and are not in this export, so they are not in the listing or the administrators list, and account admins hold a separate role, not membership of the workspace `admins` group. Its audit log carries sign-in events under the `accounts` service (`samlLogin`, `oidcBrowserLogin`, `tokenLogin` and others; on Azure Databricks also `aadBrowserLogin` and `aadTokenLogin`, Azure Databricks: audit log reference (read 2026-11-01)). The script's list of sign-in actions is a setting, `DATABRICKS_LOGIN_ACTIONS`, and an `accounts` action that looks like a sign-in but is not on it fails an input check. Creating a token is logged as `generateDbToken`, with the parameters `tokenExpirationTime`, `tokenCreatedBy`, `tokenHash` and `userId` (Databricks: audit log reference (read 2026-11-01)).

The eleven files

File	Columns the script reads	Defined in
`snowflake_users.csv`	NAME, LOGIN_NAME, EMAIL, TYPE, DISABLED, CREATED_ON, DELETED_ON, HAS_PASSWORD, HAS_MFA, HAS_RSA_PUBLIC_KEY, LAST_SUCCESS_LOGIN, OWNER	Snowflake: USERS (read 2026-11-01)
`snowflake_login_history.csv`	USER_NAME, EVENT_TIMESTAMP, IS_SUCCESS, FIRST_AUTHENTICATION_FACTOR, SECOND_AUTHENTICATION_FACTOR	Snowflake: LOGIN_HISTORY (read 2026-11-01)
`snowflake_grants_to_users.csv`	CREATED_ON, DELETED_ON, ROLE, GRANTEE_NAME, GRANTED_BY	Snowflake: GRANTS_TO_USERS (read 2026-11-01)
`databricks_scim_users.csv`	userName, active, groups (the group names of each user, separated by semicolons: our flattening of the SCIM groups list)	Databricks: list users (SCIM) (read 2026-11-01)
`databricks_audit.csv`	event_time, user_identity, service_name, action_name, request_params, response (user_identity, request_params and response as JSON text; service_name and action_name as plain text)	Databricks: audit log system table (read 2026-11-01); Databricks: audit log reference (read 2026-11-01)
`redshift_svv_user_info.csv`	user_name, user_id, superuser, external_user_id	Amazon Redshift: SVV_USER_INFO (read 2026-11-01)
`redshift_sys_connection_log.csv`	event, record_time, user_name, auth_method	Amazon Redshift: SYS_CONNECTION_LOG (read 2026-11-01)
`hr_people.csv`	email and last_day: one row per person, last_day written YYYY-MM-DD and empty while the person works here	yours (the two column names are settings)
`login_map.csv`	platform, login, email: one row for every login that is not itself an email address and has no email in the platform's own record	yours
`service_owners.csv`	platform, login, owner_email: one row for every service account; an empty owner is flagged	yours

File	Columns the script reads	Defined in
`approved_admins.csv`	platform, login: who your organization approved as an administrator, as the platform names the login	yours

What each platform keeps decides what you can show. Snowflake's LOGIN_HISTORY covers the last 365 days (one year), so a longer span needs your own copy of it (Snowflake: LOGIN_HISTORY (read 2026-11-01)). The USERS page gives no retention period of its own; it says DELETED_ON is the time the account was deleted (Snowflake: USERS (read 2026-11-01)). We read Databricks' and Redshift's user lists as the users that exist on the day you export; we found no deleted-user history on either page, so a removed account may not be in them. We found no retention period on the SYS_CONNECTION_LOG page (Amazon Redshift: SYS_CONNECTION_LOG (read 2026-11-01)), so the day-by-day check below, not a documented figure, is what tells you whether your export reaches back far enough.

Step 2. Write your policy and your lists, and read every timestamp strictly

Three settings have no default, and the script stops and says so if one is unset, because a value you never chose would be ours, not yours: `ADMIN_ROLES` (which Snowflake roles are administrator rights), `POLICY` (which sign-in methods your policy allows, per platform and for a person or a service account), and `MAX_TOKEN_DAYS` (how long your policy lets a Databricks personal access token last). The method strings in the demo policy are examples: Snowflake's page does not list the values of its two factor columns, and Redshift's does not list the values of `auth_method`, so list your own with the sign-in list below before you write the policy. A Databricks method is the audit log's action name. Timestamps are read as `YYYY-MM-DD HH:MM:SS` in UTC (a `T` in place of the space and fractional seconds are also accepted), with an optional `Z`, `+00:00` or `+0000` suffix; any other form is counted and fails an input check, never dropped. The settings you change sit at the top of the script:

```
# ---- CHANGE THESE FOR YOUR OWN DATA -------------------------------------------------------
PERIOD_START = "2026-07-01"          # first day of the review period (UTC, inclusive)
PERIOD_END = "2026-09-30"            # last day of the review period (UTC, inclusive)
AS_OF = "2026-10-15"                 # the day you took the exports (UTC); records must run to here
EXPORT_DIR = None                    # folder holding the export files named in FILES below
PLATFORMS_EXPORTED = ["snowflake", "databricks", "redshift"]   # a platform not listed is NOT
CHECKED
ADMIN_ROLES = None                   # no default: Snowflake roles that are administrator rights,
e.g. ["ACCOUNTADMIN", "SECURITYADMIN"]
DATABRICKS_ADMIN_GROUPS = ["admins"] # SCIM group names whose members are administrators
POLICY = None                        # no default: your sign-in policy, {platform: {"person":
[methods], "service": [methods]}}
MAX_TOKEN_DAYS = None                # no default: longest lifetime your policy allows a Databricks
personal access token
```

```
KNOWN_GAPS = {}                       # days you can explain, e.g. {"redshift": ["2026-08-19"]}
HR_EMAIL_COL = "email"                # HR file: the work email
HR_LAST_DAY_COL = "last_day"          # HR file: the last day of work as YYYY-MM-DD, empty while the
person works here
REDSHIFT_SIGNIN_EVENT = "authenticated"   # the SYS_CONNECTION_LOG event that is a successful sign-
in
DATABRICKS_LOGIN_ACTIONS = ["login", "samlLogin", "oidcBrowserLogin", "tokenLogin", "jwtLogin",
"mfaLogin", "aadBrowserLogin",
                            "aadTokenLogin", "multiFactorAuthenticationLogin",
"oidcTokenAuthorization", "certLogin",
                            "workspaceLoginCodeAuthentication", "accountLoginCodeAuthentication",
                            "accountlessToAccountLoginAuthentication",
"accountInHouseOAuthClientAuthentication"]   # accounts-service actions that are a sign-in
# ------------------------------------------------------------------------------------------
```

Step 3. What each list does

The user listing has one row per account that existed at any time in the period. Snowflake rows run from CREATED_ON to DELETED_ON. We treat a blank or `PERSON` type as a person and any other Snowflake type as a service account (the USERS page names `SERVICE` and `SNOWFLAKE_SERVICE` among its types and links the full list; this rule is ours, so check your own TYPE values; Snowflake: USERS (read 2026-11-01)). Any login in your owners file is a service account, on every platform. A person is matched to HR by the login map, then by a login that is itself an email, then by the email the platform holds. The flags: NO PERSON (no HR match), LEFT BUT ENABLED (an HR leaver whose account is still enabled at `AS_OF`: the timing is the SOX guide's), SERVICE NO OWNER (no owner, or an owner HR does not know or who has left).

The completeness check is the part to show first. The script takes every login seen in a successful sign-in during the period and looks for it in the listing. One that is not there is LOGIN NOT IN LISTING: on Snowflake, whose USERS view keeps deleted accounts with their DELETED_ON, the likely cause is a filtered export; on Databricks and Redshift it can also be an account removed since, and on Databricks a service principal, which the users export does not hold. On Databricks and Redshift, whose lists show only today's accounts, a login that HR says has left is printed as LEFT, NO LONGER LISTED, which is not counted as an exception. The check is this code:

```
    seen_logins = sorted({(p, l) for p, l, _, _ in signins if (p, l) not in by_login})
    for p, l in seen_logins:
        who_ = person_of(p, l, "")
        status = "LEFT, NO LONGER LISTED" if p != "snowflake" and who_ in leaver else "LOGIN NOT IN
LISTING"
        rows1.append((p, l, "(not in listing)", who_ or "", "", n_in[(p, l)], status))
```

The administrators list is who holds an administrator right at the end of `AS_OF`: a Snowflake role named in `ADMIN_ROLES` (GRANTS_TO_USERS rows still open), a Redshift superuser, or an active member of the Databricks `admins` group, compared with your approved list (NOT ON ADMIN LIST). A second list gives the Snowflake administrator grants made in the period with their date and the

granting role; GRANTED_BY is a role, not a person (Snowflake: GRANTS_TO_USERS (read 2026-11-01)).

The sign-in list counts every successful sign-in in the period per login and method (Snowflake IS_SUCCESS is yes, a Databricks status of 200, a Redshift `authenticated` event) and compares the method with the ones your policy allows for that account type: OUTSIDE POLICY. A Databricks token created in the period with no expiry, or one lasting longer than `MAX_TOKEN_DAYS`, is OUTSIDE POLICY too; the audit log also records tokens the service itself generates under the same action, so check who created each. A sign-in or a token after the person's HR last day is AFTER LAST DAY, whether or not the account is still listed.

The code

The complete download, `soc2_user_access_evidence.py`, also at https://www.datameaning.com/guides/soc-2-user-access-evidence-for-snowflake-databricks-redshift/soc2_user_access_evidence.py.

```
#!/usr/bin/env python3
"""SOC 2 user access evidence from Snowflake, Databricks and Redshift, in DuckDB.

    python3 soc2_user_access_evidence.py --demo            # clean generated records
    python3 soc2_user_access_evidence.py --demo --plant    # generated records with planted faults
    python3 soc2_user_access_evidence.py                   # your exports: set the settings below
first
    python3 soc2_user_access_evidence.py --strict          # exit 1 unless nothing needs follow-up

Reads each platform's own records, exported as CSV with the platform's published column names, plus
an HR
file, a login map, a service-account owner list, an approved-administrator list and YOUR
authentication
policy (the settings below). Prints three lists a user access review can draw on, each with its
period,
filters, run time, row count and a SHA-256 of its rows:
  1. the user listing: every account in each platform's export during the period (Databricks and
Redshift:
     the accounts that exist at the export), flagged NO PERSON, LEFT BUT ENABLED or SERVICE NO
OWNER, and any
     login seen signing in that is not in the listing (LOGIN NOT IN LISTING)
  2. the administrators at AS_OF against your approved list (NOT ON ADMIN LIST), and the admin
grants made
     in the period where the platform records them
  3. every successful sign-in in the period by method, against the methods your policy allows for
that
     account type (OUTSIDE POLICY), every Databricks token created in the period against your
expiry limit,
     and any sign-in or token after the person's HR last day (AFTER LAST DAY)
Before it builds them it runs input checks and marks the lists INCOMPLETE if one fails. A platform
you do
not export is printed as NOT CHECKED, and the result line names it.

The demo records are GENERATED (seed 230): not a company's data. Column names come from:
```

```
  Snowflake  https://docs.snowflake.com/en/sql-reference/account-usage/users
             https://docs.snowflake.com/en/sql-reference/account-usage/login_history
             https://docs.snowflake.com/en/sql-reference/account-usage/grants_to_users
  Databricks https://docs.databricks.com/api/workspace/users/list
             https://docs.databricks.com/aws/en/admin/system-tables/audit-logs
             https://docs.databricks.com/aws/en/admin/account-settings/audit-logs
             https://learn.microsoft.com/en-us/azure/databricks/admin/account-settings/audit-logs
(Entra ID sign-ins)
  Redshift   https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_USER_INFO.html
             https://docs.aws.amazon.com/redshift/latest/dg/SYS_CONNECTION_LOG.html
Our assumptions, to check against one of your own rows: Snowflake LOGIN_HISTORY.USER_NAME is
USERS.NAME; the
Databricks audit rows keep the signing-in user in request_params "user" and a token's expiry in
generateDbToken's tokenExpirationTime as Unix milliseconds (empty or 0 = no expiry); the
authentication
method strings in the demo policy are examples, so list your own with the sign-in list below.
"""
import csv
import datetime as dt
import hashlib
import json
import pathlib
import random
import re
import sys

import duckdb

# ---- CHANGE THESE FOR YOUR OWN DATA -------------------------------------------------------
PERIOD_START = "2026-07-01"          # first day of the review period (UTC, inclusive)
PERIOD_END = "2026-09-30"            # last day of the review period (UTC, inclusive)
AS_OF = "2026-10-15"                 # the day you took the exports (UTC); records must run to here
EXPORT_DIR = None                    # folder holding the export files named in FILES below
PLATFORMS_EXPORTED = ["snowflake", "databricks", "redshift"]   # a platform not listed is NOT
CHECKED
ADMIN_ROLES = None                   # no default: Snowflake roles that are administrator rights,
e.g. ["ACCOUNTADMIN", "SECURITYADMIN"]
DATABRICKS_ADMIN_GROUPS = ["admins"] # SCIM group names whose members are administrators
POLICY = None                        # no default: your sign-in policy, {platform: {"person":
[methods], "service": [methods]}}
MAX_TOKEN_DAYS = None                # no default: longest lifetime your policy allows a Databricks
personal access token
KNOWN_GAPS = {}                      # days you can explain, e.g. {"redshift": ["2026-08-19"]}
HR_EMAIL_COL = "email"               # HR file: the work email
HR_LAST_DAY_COL = "last_day"         # HR file: the last day of work as YYYY-MM-DD, empty while the
person works here
REDSHIFT_SIGNIN_EVENT = "authenticated"   # the SYS_CONNECTION_LOG event that is a successful sign-
in
DATABRICKS_LOGIN_ACTIONS = ["login", "samlLogin", "oidcBrowserLogin", "tokenLogin", "jwtLogin",
"mfaLogin", "aadBrowserLogin",
                            "aadTokenLogin", "multiFactorAuthenticationLogin",
"oidcTokenAuthorization", "certLogin",
                            "workspaceLoginCodeAuthentication", "accountLoginCodeAuthentication",
                            "accountlessToAccountLoginAuthentication",
"accountInHouseOAuthClientAuthentication"]   # accounts-service actions that are a sign-in
# -------------------------------------------------------------------------------------------

FILES = {
```

```
    "snowflake_users": "snowflake_users.csv",
    "snowflake_login_history": "snowflake_login_history.csv",
    "snowflake_grants_to_users": "snowflake_grants_to_users.csv",
    "databricks_scim_users": "databricks_scim_users.csv",
    "databricks_audit": "databricks_audit.csv",
    "redshift_svv_user_info": "redshift_svv_user_info.csv",
    "redshift_sys_connection_log": "redshift_sys_connection_log.csv",
    "hr_people": "hr_people.csv",
    "login_map": "login_map.csv",
    "service_owners": "service_owners.csv",
    "approved_admins": "approved_admins.csv",
}
PLATFORM_FILES = {
    "snowflake": ["snowflake_users", "snowflake_login_history", "snowflake_grants_to_users"],
    "databricks": ["databricks_scim_users", "databricks_audit"],
    "redshift": ["redshift_svv_user_info", "redshift_sys_connection_log"],
}
COLUMNS = {   # the platforms' published column names, lower-cased here; the file's own case does
not matter
    "snowflake_users": ["name", "login_name", "email", "type", "disabled", "created_on",
"deleted_on", "has_password",
                        "has_mfa", "has_rsa_public_key", "last_success_login", "owner"],
    "snowflake_login_history": ["user_name", "event_timestamp", "is_success",
"first_authentication_factor",
                                "second_authentication_factor"],
    "snowflake_grants_to_users": ["created_on", "deleted_on", "role", "grantee_name",
"granted_by"],
    "databricks_scim_users": ["username", "active", "groups"],
    "databricks_audit": ["event_time", "user_identity", "service_name", "action_name",
"request_params", "response"],
    "redshift_svv_user_info": ["user_name", "user_id", "superuser", "external_user_id"],
    "redshift_sys_connection_log": ["event", "record_time", "user_name", "auth_method"],
    "login_map": ["platform", "login", "email"],
    "service_owners": ["platform", "login", "owner_email"],
    "approved_admins": ["platform", "login"],
}
LOGIN_LIKE = re.compile(r"login|authentication$", re.I)   # an accounts action that looks like a
sign-in (not ...UpdateUserAuthPolicy)
OUT_DIR = pathlib.Path("evidence-out")
GEN_DIR = pathlib.Path("generated")
TS_RE = "\\d{4}-\\d{2}-\\d{2}[ T]\\d{2}:\\d{2}:\\d{2}(\\.\\d+)?( ?(Z|\\+00:00|\\+0000))?"
DATE_RE = "\\d{4}-\\d{2}-\\d{2}"
PLATFORMS = ["snowflake", "databricks", "redshift"]
DEMO_POLICY = {
    "snowflake": {"person": ["SAML2_ASSERTION", "PASSWORD + DUO_PUSH"], "service":
["RSA_KEYPAIR"]},
    "databricks": {"person": ["samlLogin", "oidcBrowserLogin"], "service": ["tokenLogin"]},
    "redshift": {"person": ["IAM"], "service": ["IAM", "password"]},
}


def ts(col):
    """A timestamp column read strictly: YYYY-MM-DD HH:MM:SS, taken as UTC (a Z, +00:00 or +0000
suffix is
    accepted, any other offset is NULL and counted). Export from a session set to UTC."""
    return (f"CASE WHEN regexp_full_match(trim({col}), '{TS_RE}') THEN "
            f"try_strptime(substr(replace(trim({col}), 'T', ' '), 1, 19), '%Y-%m-%d %H:%M:%S')
END")
```

```
def day(col):
    return (f"CASE WHEN regexp_full_match(trim({col}), '{DATE_RE}') THEN "
            f"try_strptime(trim({col}), '%Y-%m-%d')::DATE END")


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()


# ---------------------------------------------------------------------------------------------
# Demo records: generated, in each platform's published column names
# ---------------------------------------------------------------------------------------------
def generate(plant):
    rnd = random.Random(230)
    out = GEN_DIR / ("planted" if plant else "clean")
    out.mkdir(parents=True, exist_ok=True)
    D = dt.date
    start, as_of = D(2026, 7, 1), D(2026, 10, 15)
    em = lambda k: f"{k}@example.test"
    rows = {k: [] for k in FILES}

    def at(d, hour=None):
        return dt.datetime.combine(d, dt.time(hour if hour is not None else rnd.randrange(8, 18),
rnd.randrange(60), rnd.randrange(60)))

    sfts = lambda t: t.strftime("%Y-%m-%d %H:%M:%S.000 +0000")
    plain = lambda t: t.strftime("%Y-%m-%d %H:%M:%S")

    def every(a, b, step, fn):
        d = a
        while d <= min(b, as_of):
            fn(at(d))
            d += dt.timedelta(days=step)

    # ---- HR, login map, owners, approved administrators
    rows["hr_people"] = [[em(k), ""] for k in ("p01", "p02", "p03", "p04", "p05")] + [[em("p06"),
"2026-08-07"]]
    if plant:
        rows["hr_people"].append([em("p07"), "8/5/26"])   # a date that is not YYYY-MM-DD
    rows["login_map"] = [["redshift", k, em(p)] for k, p in (("ana", "p01"), ("ben", "p02"), ("cy",
"p03"), ("dee", "p04"), ("eli", "p06"))]
    rows["service_owners"] = [["snowflake", "svc_etl", em("p02")], ["databricks", "svc-
jobs@example.test", em("p02")]]
    rows["service_owners"].append(["redshift", "etl_loader", "" if plant else em("p02")])   #
planted: a service account with no owner
    rows["approved_admins"] = [["snowflake", "p01"], ["snowflake", "p02"], ["databricks",
em("p01")], ["redshift", "ana"]]

    # ---- Snowflake
    sf_logins = []   # (user, time, success, first, second)

    def sf_login(t, user, ok="YES", first="SAML2_ASSERTION", second=""):
        sf_logins.append((user, t, ok, first, second))

    every(start, as_of, 1, lambda t: sf_login(t, "SVC_ETL", first="RSA_KEYPAIR"))
```

```
    every(D(2026, 7, 1), as_of, 3, lambda t: sf_login(t, "P01"))
    every(D(2026, 7, 2), as_of, 4, lambda t: sf_login(t, "P02"))
    every(D(2026, 7, 3), as_of, 5, lambda t: sf_login(t, "P03", first="PASSWORD" if plant and
t.date() >= D(2026, 9, 10) else "SAML2_ASSERTION"))
    # planted: from 10 September P03 signs in with a password and no second factor
    every(D(2026, 7, 4), as_of, 6, lambda t: sf_login(t, "P04"))
    every(D(2026, 7, 1), D(2026, 8, 7), 5, lambda t: sf_login(t, "P06"))
    sf_login(at(D(2026, 8, 3)), "P03", ok="NO", first="PASSWORD")        # a refused attempt is not
a sign-in
    if plant:
        every(D(2026, 8, 1), as_of, 9, lambda t: sf_login(t, "SF_GHOST"))   # signs in, but is not
in the users export
    last_ok = {}
    for u, t, ok, _, _ in sf_logins:
        if ok == "YES":
            last_ok[u] = max(last_ok.get(u, t), t)
    users = [("P01", "PERSON", em("p01")), ("P02", "PERSON", em("p02")), ("P03", "PERSON",
em("p03")), ("P04", "PERSON", em("p04")),
             ("P06", "PERSON", em("p06")), ("SVC_ETL", "SERVICE", em("p02"))]
    if plant:
        users.append(("CONTRACTOR1", "PERSON", ""))
        last_ok["CONTRACTOR1"] = at(D(2026, 9, 1))
        sf_logins.append(("CONTRACTOR1", last_ok["CONTRACTOR1"], "YES", "SAML2_ASSERTION", ""))
    for u, typ, mail in users:
        left = u == "P06" and not plant
        rows["snowflake_users"].append([u, u, mail.upper() if mail else "", typ, "true" if left
else "false", sfts(at(D(2026, 3, 2))),
                                        sfts(at(D(2026, 8, 8))) if left else "", "false" if typ ==
"SERVICE" else "true",
                                        "true" if typ == "PERSON" else "false", "true" if typ ==
"SERVICE" else "false",
                                        sfts(last_ok[u]) if u in last_ok else "", "USERADMIN"])
    if plant:   # P04's later sign-ins are missing from the login export, though USERS still knows
the last one
        sf_logins = [x for x in sf_logins if not (x[0] == "P04" and x[1].date() >= D(2026, 9, 20))]
    for u, t, ok, a, b in sf_logins:
        rows["snowflake_login_history"].append([u, sfts(t), ok, a, b])
    g =rows["snowflake_grants_to_users"]
    g.append([sfts(at(D(2026, 1, 5))), "", "ACCOUNTADMIN", "P01", "ACCOUNTADMIN"])
    g.append([sfts(at(D(2026, 8, 12))), "", "SECURITYADMIN", "P02", "ACCOUNTADMIN"])
    g.append([sfts(at(D(2026, 3, 2))), "", "ANALYST_ROLE", "P03", "USERADMIN"])
    if plant:
        g.append([sfts(at(D(2026, 9, 2))), "", "ACCOUNTADMIN", "P03", "ACCOUNTADMIN"])

    # ---- Databricks
    def dbx(t, actor, service, action, params, status=200):
        rows["databricks_audit"].append([plain(t), json.dumps({"email": actor}), service, action,
json.dumps(params),
                                         json.dumps({"statusCode": status})])

    def dbx_login(t, user, action):
        dbx(t, user, "accounts", action, {"user": user, "authenticationMethod": "demo"})

    scim = [(em("p01"), "true", "admins;data-eng"), (em("p02"), "true", "data-eng"), (em("p03"),
"true", "data-eng"),
            (em("p04"), "true", "data-eng"), (em("p06"), "true" if plant else "false", "data-eng"),
("svc-jobs@example.test", "true", "")]
    if plant:
```

```
        scim.append(("temp.consultant@example.test", "true", ""))
        scim[2] = (em("p03"), "true", "admins;data-eng")
    rows["databricks_scim_users"] = [list(x) for x in scim]
    d = start
    while d <= as_of:
        dbx(at(d), em("p01"), "unityCatalog", "getTable", {"full_name_arg": "demo.records.t"})
        d += dt.timedelta(days=1)
    every(D(2026, 7, 1), as_of, 4, lambda t: dbx_login(t, em("p01"), "samlLogin"))
    every(D(2026, 7, 2), as_of, 5, lambda t: dbx_login(t, em("p02"), "oidcBrowserLogin"))
    every(D(2026, 7, 3), as_of, 6, lambda t: dbx_login(t, em("p03"), "samlLogin"))
    every(D(2026, 7, 4), as_of, 7, lambda t: dbx_login(t, em("p04"), "samlLogin"))
    every(D(2026, 7, 1), D(2026, 8, 7), 6, lambda t: dbx_login(t, em("p06"), "samlLogin"))
    every(D(2026, 7, 1), as_of, 1, lambda t: dbx(t, "svc-jobs@example.test", "accounts",
"tokenLogin",
                                                 {"tokenId": "demo-token", "user": "svc-
jobs@example.test", "authenticationMethod": "demo"}))
    dbx(at(D(2026, 8, 5)), em("p03"), "accounts", "samlLogin", {"user": em("p03")}, status=401)   #
refused

    def dbx_token(t, who_, days):
        exp = "" if days is None else str(int((t +
dt.timedelta(days=days)).replace(tzinfo=dt.timezone.utc).timestamp() * 1000))
        dbx(t, who_, "accounts", "generateDbToken", {"tokenExpirationTime": exp, "tokenCreatedBy":
who_, "tokenHash": "demo-hash", "userId": "1"})

    dbx_token(at(D(2026, 8, 20)), em("p03"), 30)
    dbx_token(at(D(2026, 9, 1)), "svc-jobs@example.test", 90)
    if plant:
        dbx_token(at(D(2026, 9, 15)), em("p02"), None)                      # no expiry
        dbx_token(at(D(2026, 9, 20)), "svc-jobs@example.test", 365)         # longer than the
policy allows
        dbx_login(at(D(2026, 9, 25)), em("p04"), "tokenLogin")              # a person signing in
with a token

    # ---- Redshift
    rsu = [(101, "ana", "t", em("p01")), (102, "ben", "f", em("p02")), (103, "cy", "f", em("p03")),
(104, "dee", "f", em("p04")),
           (200, "etl_loader", "f", "")]
    if plant:
        rsu += [(105, "eli", "f", em("p06"))]
        rsu[3] = (104, "dee", "t", em("p04"))   # a superuser nobody approved
    rows["redshift_svv_user_info"] = [[b, a, c, e] for a, b, c, e in rsu]

    def rs_sign(t, user, method="IAM"):
        rows["redshift_sys_connection_log"].append(["authenticated", plain(t), user, method])
        rows["redshift_sys_connection_log"].append(["initiating session", plain(t +
dt.timedelta(seconds=1)), user, method])

    every(start, as_of, 1, lambda t: rs_sign(t, "etl_loader", "password"))
    every(D(2026, 7, 1), as_of, 3, lambda t: rs_sign(t, "ana"))
    every(D(2026, 7, 2), as_of, 4, lambda t: rs_sign(t, "ben"))
    every(D(2026, 7, 3), as_of, 5, lambda t: rs_sign(t, "cy", "password" if plant and t.date() >=
D(2026, 9, 1) else "IAM"))
    every(D(2026, 7, 4), as_of, 6, lambda t: rs_sign(t, "dee"))
    every(D(2026, 7, 1), D(2026, 8, 7), 5, lambda t: rs_sign(t, "eli"))
    if plant:
        rs_sign(at(D(2026, 8, 20)), "eli")   # a former employee signs in after the last day
    rows["redshift_sys_connection_log"].append(["authentication failure", plain(at(D(2026, 8, 9))),
```

```
"ben", "password"])
    if plant:
        rows["redshift_sys_connection_log"] = [r for r in rows["redshift_sys_connection_log"] if
not r[1].startswith("2026-08-19")]

    # ---- write
    for key, cs in COLUMNS.items():
        if key in ("login_map", "service_owners", "approved_admins"):
            continue
        head = [c.upper() for c in cs] if key.startswith("snowflake") else cs
        if key == "databricks_scim_users":
            head = ["userName", "active", "groups"]
        with open(out / FILES[key], "w", newline="") as f:
            w = csv.writer(f)
            w.writerow(head)
            w.writerows(rows[key])
    for key, head in [("hr_people", ["email", "last_day"]), ("login_map", ["platform", "login",
"email"]),
                      ("service_owners", ["platform", "login", "owner_email"]), ("approved_admins",
["platform", "login"])]:
        with open(out / FILES[key], "w", newline="") as f:
            w = csv.writer(f)
            w.writerow(head)
            w.writerows(rows[key])
    return str(out)


# ---------------------------------------------------------------------------------------
def load(con, key, folder, cols, renames=None):
    """Read a CSV as text with its header lower-cased; return a problem string or None."""
    path = pathlib.Path(folder) / FILES[key]
    if not path.is_file():
        return f"{FILES[key]} not found in {folder}"
    names = [r[0] for r in con.execute("DESCRIBE SELECT * FROM read_csv(?, header=true,
all_varchar=true)", [str(path)]).fetchall()]
    low = [n.strip().lower() for n in names]
    renames = renames or {}
    want = [renames.get(c, c) for c in cols]
    missing = [c for c in want if c not in low]
    if missing or len(set(low)) != len(low):
        return f"{FILES[key]} lacks column(s) {missing}" if missing else f"{FILES[key]} has a
column name twice"
    sel = ", ".join(f'"{names[low.index(c)]}" AS "{cols[i]}"' for i, c in enumerate(want))
    con.execute(f"CREATE TABLE raw_{key} AS SELECT {sel} FROM read_csv(?, header=true,
all_varchar=true)", [str(path)])
    return None


def population(name, title, filters, rows, header, complete):
    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== {title}: {name}" + ("" if complete else "   ** INCOMPLETE: see input checks **"))
    print(f"period    : {PERIOD_START} to {PERIOD_END} (UTC, inclusive); accounts and
administrators as of {AS_OF}")
    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)}")


def show(rows, flagged, status_at, limit=12):
    for r in rows:   # every row when there are few, otherwise only the ones to explain
        if len(rows) <= limit or flagged(r[status_at]):
            print("  " + " | ".join("-" if v is None or v == "" else str(v) for v in r))


def main():
    demo, plant, strict = "--demo" in sys.argv, "--plant" in sys.argv, "--strict" in sys.argv
    folder, admin_roles, policy, max_token = EXPORT_DIR, ADMIN_ROLES, POLICY, MAX_TOKEN_DAYS
    if demo:
        folder = generate(plant)
        admin_roles, policy, max_token = ["ACCOUNTADMIN", "SECURITYADMIN"], DEMO_POLICY, 90
        print(f"input     : GENERATED records (not a company's data), seed 230, plant={plant}")
    else:
        print("input     : " + str(folder))
    exported = [p for p in PLATFORMS if p in PLATFORMS_EXPORTED]
    not_checked = [p for p in PLATFORMS if p not in exported]
    if folder is None:
        print("STOP: EXPORT_DIR is not set. Point it at the folder holding your export files, then
run again.")
        return 2
    if [p for p in PLATFORMS_EXPORTED if p not in PLATFORMS] or not exported:
        print(f"STOP: PLATFORMS_EXPORTED must list snowflake, databricks and/or redshift (got
{PLATFORMS_EXPORTED}).")
        return 2
    if "snowflake" in exported and not admin_roles:
        print("STOP: ADMIN_ROLES has no default. Name the Snowflake roles that are administrator
rights, then run again.")
        return 2
    if not isinstance(policy, dict) or not policy:
        print("STOP: POLICY has no default. Write down which sign-in methods your policy allows,
per platform and account type, then run again.")
        return 2
    if "databricks" in exported and not (isinstance(max_token, int) and max_token >= 1):
        print("STOP: MAX_TOKEN_DAYS has no default. Say how many days a personal access token may
last under your policy, then run again.")
        return 2
    bad_settings = [n for n, v in [("PERIOD_START", PERIOD_START), ("PERIOD_END", PERIOD_END),
("AS_OF", AS_OF)] if not re.fullmatch(DATE_RE, str(v))]
    if bad_settings or not PERIOD_START <= PERIOD_END <= AS_OF:
        print("STOP: PERIOD_START, PERIOD_END and AS_OF must be YYYY-MM-DD with PERIOD_START <=
PERIOD_END <= AS_OF.")
        return 2
    p0, p1, asof = PERIOD_START, PERIOD_END, AS_OF
    p1_end = (dt.date.fromisoformat(p1) + dt.timedelta(days=1)).isoformat()
    asof_end = (dt.date.fromisoformat(asof) + dt.timedelta(days=1)).isoformat()
    print("platforms : checked " + ", ".join(exported) + ("; NOT CHECKED: " + ",
".join(not_checked) if not_checked else ""))
    pol = {(p.lower(), t.lower()): {m.strip().lower() for m in ms} for p, d in policy.items() for
t, ms in d.items()}
    con = duckdb.connect()

    # ---- load the files; a missing file or column stops the run: a check that cannot run is a
```

```
failure
    problems = []
    for p in exported:
        for key in PLATFORM_FILES[p]:
            problems.append(load(con, key, folder, COLUMNS[key]))
    for key in ("login_map", "service_owners", "approved_admins"):
        problems.append(load(con, key, folder, COLUMNS[key]))
    problems.append(load(con, "hr_people", folder, ["email", "last_day"], {"email":
HR_EMAIL_COL.lower(), "last_day": HR_LAST_DAY_COL.lower()}))
    problems = [x for x in problems if x]
    if problems:
        print("\n== Input checks")
        for x in problems:
            print("[FAIL] " + x)
        print("RESULT: STOPPED: the files above could not be read, so no list was built")
        return 2

    checks = []   # (label, number of rows at fault, what the number counts)

    def check(label, n, what):
        checks.append((label, n, what))

    # ---- the lists you keep: HR, login map, service owners, approved administrators
    con.execute(f"""CREATE TABLE hr AS SELECT lower(trim(coalesce(email, ''))) AS email,
trim(coalesce(last_day, '')) AS raw,
                   {day('last_day')} AS last_day FROM raw_hr_people""")
    check("HR file: rows with no email", con.execute("SELECT count(*) FROM hr WHERE email =
''").fetchone()[0], "rows")
    check("HR file: last days not written as YYYY-MM-DD", con.execute("SELECT count(*) FROM hr
WHERE raw <> '' AND last_day IS NULL").fetchone()[0], "rows")
    hr = {}   # email -> (still works here, last day)
    for email, cur, last in con.execute("SELECT email, bool_or(raw = ''), max(last_day) FROM hr
WHERE email <> '' GROUP BY email ORDER BY email").fetchall():
        hr[email] = (bool(cur), last)
    leaver = {e: v[1] for e, v in hr.items() if not v[0] and v[1] is not None and v[1].isoformat()
<= asof}
    ident = {(p, l): e for p, l, e in con.execute("""SELECT lower(trim(platform)),
lower(trim(login)), lower(trim(email)) FROM raw_login_map
                   WHERE trim(coalesce(login, '')) <> '' AND trim(coalesce(email, '')) <> '' ORDER
BY 1, 2, 3""").fetchall()}
    check("login map: rows with no login or no email", con.execute(
        "SELECT count(*) FROM raw_login_map WHERE trim(coalesce(login, '')) = '' OR
trim(coalesce(email, '')) = ''").fetchone()[0], "rows")
    owners = {(p, l): o for p, l, o in con.execute("""SELECT lower(trim(platform)),
lower(trim(login)), lower(trim(coalesce(owner_email, '')))
                   FROM raw_service_owners WHERE trim(coalesce(login, '')) <> '' ORDER BY 1, 2,
3""").fetchall()}
    approved = {(p, l) for p, l in con.execute("""SELECT lower(trim(platform)), lower(trim(login))
FROM raw_approved_admins
                   WHERE trim(coalesce(login, '')) <> '' """).fetchall()}

    accts = []        # platform, login, hint (email the platform holds), type, enabled at AS_OF,
in the period, superuser, groups
    signins = []      # platform, login, method, time
    tokens = []       # login, time, lifetime in days or None
    all_events = {}   # platform -> set of days with any record in the sign-in source
    gaps = {}
    admin_rows = []   # platform, login, how, since
```

```
    admin_grants = [] # Snowflake admin grants made in the period

    # ---- Snowflake
    if "snowflake" in exported:
        con.execute(f"""CREATE TABLE sf_u AS SELECT lower(trim(coalesce(name, ''))) AS login,
lower(trim(coalesce(email, ''))) AS email,
               upper(trim(coalesce(type, ''))) AS type, lower(replace(trim(coalesce(disabled, '')),
'"', '')) AS disabled,
               {ts('created_on')} AS created, {ts('deleted_on')} AS deleted,
trim(coalesce(deleted_on, '')) AS deleted_text,
               {ts('last_success_login')} AS last_ok, trim(coalesce(last_success_login, '')) AS
last_ok_text FROM raw_snowflake_users""")
        con.execute(f"""CREATE TABLE sf_l AS SELECT lower(trim(coalesce(user_name, ''))) AS login,
{ts('event_timestamp')} AS ts,
               CASE WHEN lower(trim(is_success)) IN ('yes', 'true') THEN true WHEN
lower(trim(is_success)) IN ('no', 'false') THEN false END AS ok,
               upper(trim(coalesce(first_authentication_factor, ''))) AS first_f,
upper(trim(coalesce(second_authentication_factor, ''))) AS second_f
               FROM raw_snowflake_login_history""")
        con.execute(f"""CREATE TABLE sf_g AS SELECT {ts('created_on')} AS created,
{ts('deleted_on')} AS deleted, trim(coalesce(deleted_on, '')) AS deleted_text,
               lower(trim(coalesce(role, ''))) AS role, lower(trim(coalesce(grantee_name, ''))) AS
login, lower(trim(coalesce(granted_by, ''))) AS granted_by
               FROM raw_snowflake_grants_to_users""")
        check("USERS: empty export", int(con.execute("SELECT count(*) FROM sf_u").fetchone()[0] ==
0), "export")
        check("USERS: rows with no NAME", con.execute("SELECT count(*) FROM sf_u WHERE login =
''").fetchone()[0], "rows")
        check("USERS: CREATED_ON not written as YYYY-MM-DD HH:MM:SS", con.execute("SELECT count(*)
FROM sf_u WHERE created IS NULL").fetchone()[0], "rows")
        check("USERS: DELETED_ON or LAST_SUCCESS_LOGIN not written as YYYY-MM-DD HH:MM:SS",
con.execute(
            "SELECT count(*) FROM sf_u WHERE (deleted_text <> '' AND deleted IS NULL) OR
(last_ok_text <> '' AND last_ok IS NULL)").fetchone()[0], "rows")
        check("USERS: DISABLED is not true or false", con.execute("SELECT count(*) FROM sf_u WHERE
disabled NOT IN ('', 'true', 'false')").fetchone()[0], "rows")
        check("LOGIN_HISTORY: EVENT_TIMESTAMP or IS_SUCCESS unreadable, or a successful sign-in
with no USER_NAME", con.execute(
            "SELECT count(*) FROM sf_l WHERE ts IS NULL OR ok IS NULL OR (ok AND login =
'')").fetchone()[0], "rows")
        check("GRANTS_TO_USERS: dates unreadable", con.execute(
            "SELECT count(*) FROM sf_g WHERE created IS NULL OR (deleted_text <> '' AND deleted IS
NULL)").fetchone()[0], "rows")
        wanted = {r.lower() for r in admin_roles}
        seen_roles = {r[0] for r in con.execute("SELECT DISTINCT role FROM sf_g").fetchall()}
        check("ADMIN_ROLES: a name that matches no grant in GRANTS_TO_USERS (a typo hides every
holder)",
              len(wanted - seen_roles), "names: " + ", ".join(sorted(wanted - seen_roles)) if
wanted - seen_roles else "names")
        # the sign-in records must agree with the users export: LAST_SUCCESS_LOGIN names a day with
a sign-in
        check("LOGIN_HISTORY: a user's LAST_SUCCESS_LOGIN day has no successful sign-in on it
(filtered or short export)", con.execute(f"""
              SELECT count(*) FROM sf_u u WHERE u.last_ok >= TIMESTAMP '{p0}' AND u.last_ok <
TIMESTAMP '{asof}' AND NOT EXISTS (
                SELECT 1 FROM sf_l l WHERE l.login = u.login AND l.ok AND l.ts::DATE =
u.last_ok::DATE)""").fetchone()[0], "users")
        all_events["snowflake"] = {r[0] for r in con.execute("SELECT DISTINCT ts::DATE FROM sf_l
```

```
WHERE ts IS NOT NULL").fetchall()}
        for login, email, typ, dis, cr, dl, last in con.execute(f"""SELECT login, email, type,
disabled, created, deleted, last_ok FROM sf_u
             WHERE login <> '' AND created IS NOT NULL ORDER BY login, created""").fetchall():
            gone = dl is not None and dl.isoformat(sep=" ") < asof_end
            started = cr.isoformat(sep=" ") < p1_end
            in_p = started and (dl is None or dl.date().isoformat() >= p0)
            enabled = started and not gone and dis != "true" and cr.isoformat(sep=" ") < asof_end
            accts.append(dict(platform="snowflake", login=login, hint=email, type="service" if typ
not in ("", "PERSON") else "person",
                              enabled=enabled, in_period=in_p, superuser=False, groups=[]))
        for login, first, second, t in con.execute(f"""SELECT login, first_f, second_f, ts FROM
sf_l WHERE ok AND ts >= TIMESTAMP '{p0}'
             AND ts < TIMESTAMP '{p1_end}' ORDER BY ts, login, first_f, second_f""").fetchall():
            signins.append(("snowflake", login, (first or "(none recorded)") + (" + " + second if
second else ""), t))
        for role, login, cr, dl, by in con.execute(f"""SELECT role, login, created, deleted,
granted_by FROM sf_g WHERE role IN
             ({", ".join("'" + r.replace("'", "''") + "'" for r in sorted(wanted))}) AND created
IS NOT NULL ORDER BY created, role, login""").fetchall():
            if cr.isoformat(sep=" ") < asof_end and (dl is None or dl.isoformat(sep=" ") >=
asof_end):
                admin_rows.append(("snowflake", login, "role " + role.upper(),
cr.date().isoformat()))
            if p0 <= cr.date().isoformat() <= p1:
                admin_grants.append((cr.date().isoformat(), role.upper(), login, by.upper(), "yes"
if dl is None or dl.isoformat(sep=" ") >= asof_end else "no"))

    # ---- Databricks
    if "databricks" in exported:
        con.execute("""CREATE TABLE dbx_u AS SELECT lower(trim(coalesce(username, ''))) AS login,
lower(trim(coalesce(active, ''))) AS active,
             coalesce(groups, '') AS groups FROM raw_databricks_scim_users""")
        con.execute(f"""CREATE TABLE dbx_a AS SELECT {ts('event_time')} AS ts,
lower(trim(coalesce(service_name, ''))) AS service,
             trim(replace(coalesce(action_name, ''), '"', '')) AS action, request_params,
response,
             CASE WHEN json_valid(user_identity) THEN json_extract_string(user_identity,
'$.email') END AS actor,
             trim(coalesce(user_identity, '')) <> '' AND NOT coalesce(json_valid(user_identity),
false) AS bad_identity
             FROM raw_databricks_audit""")
        check("SCIM users: empty export", int(con.execute("SELECT count(*) FROM dbx_u").fetchone()
[0] == 0), "export")
        check("SCIM users: userName empty or listed twice", con.execute(
            "SELECT count(*) FROM dbx_u WHERE login = '' OR login IN (SELECT login FROM dbx_u GROUP
BY login HAVING count(*) > 1)").fetchone()[0], "rows")
        check("SCIM users: active is not true or false", con.execute("SELECT count(*) FROM dbx_u
WHERE active NOT IN ('true', 'false')").fetchone()[0], "rows")
        check("system.access.audit: event_time not written as YYYY-MM-DD HH:MM:SS",
con.execute("SELECT count(*) FROM dbx_a WHERE ts IS NULL").fetchone()[0], "rows")
        check("system.access.audit: user_identity that is not JSON", con.execute("SELECT count(*)
FROM dbx_a WHERE bad_identity").fetchone()[0], "rows")
        # a day counts only if the accounts service, where sign-ins are logged, has a record on it
        all_events["databricks"] = {r[0] for r in con.execute("SELECT DISTINCT ts::DATE FROM dbx_a
WHERE ts IS NOT NULL AND service = 'accounts'").fetchall()}
        login_set = {a.lower() for a in DATABRICKS_LOGIN_ACTIONS}
        unknown = sorted({r[0] for r in con.execute("SELECT DISTINCT action FROM dbx_a WHERE
```

```
service = 'accounts'").fetchall()
                          if LOGIN_LIKE.search(r[0]) and r[0].lower() not in login_set})
        check("system.access.audit: accounts actions that look like a sign-in but are not in
DATABRICKS_LOGIN_ACTIONS", len(unknown),
              "actions" + (": " + ", ".join(unknown[:5]) if unknown else ""))
        admin_groups = {g.lower() for g in DATABRICKS_ADMIN_GROUPS}
        for login, active, groups in con.execute("SELECT login, active, groups FROM dbx_u WHERE
login <> '' ORDER BY login").fetchall():
            gl = sorted({g.strip().lower() for g in groups.split(";") if g.strip()})
            accts.append(dict(platform="databricks", login=login, hint="", type="person",
enabled=active == "true", in_period=True,
                              superuser=False, groups=gl))
            if active == "true" and admin_groups & set(gl):
                admin_rows.append(("databricks", login, "group " + ", ".join(sorted(admin_groups &
set(gl))), ""))
        bad_json = bad_status = bad_tok = 0
        for t, service, action, rp, resp, actor in con.execute(f"""SELECT ts, service, action,
request_params, response, actor FROM dbx_a
              WHERE service = 'accounts' AND ts >= TIMESTAMP '{p0}' AND ts < TIMESTAMP '{p1_end}'
AND ts IS NOT NULL
              ORDER BY ts, action, request_params""").fetchall():
            is_login, is_tok = action.lower() in login_set, action == "generateDbToken"
            if not (is_login or is_tok):
                continue
            try:
                params = json.loads(rp)
                status = json.loads(resp).get("statusCode", json.loads(resp).get("status_code"))
                if not str(status).strip().isdigit():
                    bad_status += 1
                    continue
                if int(status) != 200:   # a refused attempt is not a sign-in
                    continue
                if is_login:
                    who_ = str(params.get("user") or actor or "").strip().lower()
                    if not who_:
                        bad_json += 1
                        continue
                    signins.append(("databricks", who_, action, t))
                else:
                    who_ = str(params.get("tokenCreatedBy") or "").strip().lower()
                    exp = str(params.get("tokenExpirationTime") or "").strip()
                    if not who_ or (exp and not re.fullmatch(r"-?\d+", exp)):
                        bad_tok += 1
                        continue
                    life = None
                    if exp and int(exp) > 0:
                        life = round((int(exp) / 1000 -
t.replace(tzinfo=dt.timezone.utc).timestamp()) / 86400)
                    tokens.append((who_, t, life))
            except (ValueError, KeyError, TypeError, AttributeError):
                bad_json += 1
        check("system.access.audit: sign-in rows with no readable user or request_params",
bad_json, "rows")
        check("system.access.audit: sign-in or token rows with no readable status", bad_status,
"rows")
        check("system.access.audit: token rows with no creator or an expiry that is not a number",
bad_tok, "rows")
```

```
    # ---- Redshift
    if "redshift" in exported:
        con.execute("""CREATE TABLE rs_u AS SELECT lower(trim(coalesce(user_name, ''))) AS login,
lower(trim(coalesce(superuser, ''))) AS su,
             lower(trim(coalesce(external_user_id, ''))) AS ext FROM
raw_redshift_svv_user_info""")
        con.execute(f"""CREATE TABLE rs_c AS SELECT lower(trim(coalesce(event, ''))) AS event,
{ts('record_time')} AS ts,
             lower(trim(coalesce(user_name, ''))) AS login, lower(trim(coalesce(auth_method, '')))
AS method FROM raw_redshift_sys_connection_log""")
        check("SVV_USER_INFO: empty export", int(con.execute("SELECT count(*) FROM
rs_u").fetchone()[0] == 0), "export")
        check("SVV_USER_INFO: user_name empty or listed twice", con.execute(
            "SELECT count(*) FROM rs_u WHERE login = '' OR login IN (SELECT login FROM rs_u GROUP
BY login HAVING count(*) > 1)").fetchone()[0], "rows")
        check("SVV_USER_INFO: superuser is not true or false", con.execute(
            "SELECT count(*) FROM rs_u WHERE su NOT IN ('true', 't', '1', 'false', 'f',
'0')").fetchone()[0], "rows")
        check("SYS_CONNECTION_LOG: empty export (a non-superuser sees no rows)",
int(con.execute("SELECT count(*) FROM rs_c").fetchone()[0] == 0), "export")
        check("SYS_CONNECTION_LOG: record_time not written as YYYY-MM-DD HH:MM:SS, or no user on a
sign-in",
             con.execute("SELECT count(*) FROM rs_c WHERE ts IS NULL OR (event = ? AND login =
'')", [REDSHIFT_SIGNIN_EVENT.lower()]).fetchone()[0], "rows")
        all_events["redshift"] = {r[0] for r in con.execute("SELECT DISTINCT ts::DATE FROM rs_c
WHERE ts IS NOT NULL").fetchall()}
        for login, su, ext in con.execute("SELECT login, su, ext FROM rs_u WHERE login <> '' ORDER
BY login").fetchall():
            is_su = su in ("true", "t", "1")
            accts.append(dict(platform="redshift", login=login, hint=ext, type="person",
enabled=True, in_period=True, superuser=is_su, groups=[]))
            if is_su:
                admin_rows.append(("redshift", login, "superuser", ""))
        for login, method, t in con.execute(f"""SELECT login, method, ts FROM rs_c WHERE event = ?
AND ts >= TIMESTAMP '{p0}'
              AND ts < TIMESTAMP '{p1_end}' AND login <> '' ORDER BY ts, login, method""",
[REDSHIFT_SIGNIN_EVENT.lower()]).fetchall():
            signins.append(("redshift", login, method or "(none recorded)", t))

    # ---- completeness over the whole span the lists depend on: every day from PERIOD_START to
AS_OF has a record
    d0, d1 = dt.date.fromisoformat(p0), dt.date.fromisoformat(asof)
    for p in exported:
        explained = set(KNOWN_GAPS.get(p, []))
        missing = [(d0 + dt.timedelta(days=i)).isoformat() for i in range((d1 - d0).days + 1)
                   if (d0 + dt.timedelta(days=i)) not in all_events[p]]
        gaps[p] = [m for m in missing if m not in explained]
        check(f"{p}: days with no sign-in record from {p0} to {asof}", len(gaps[p]),
              "days" + (": " + ", ".join(gaps[p][:5]) + (" ..." if len(gaps[p]) > 5 else "") if
gaps[p] else ""))
    # a key in the policy, the owners or the map with no platform behind it is a typo, not a rule
    for (p, t) in pol:
        if p not in PLATFORMS or t not in ("person", "service"):
            check("POLICY: a platform or account type this script does not know", 1, f"entry
{p}/{t}")
    ok = all(n == 0 for _, n, _ in checks)
    print("\n== Input checks")
    for label, n, what in checks:
```

```
        print(f"[{'ok' if n == 0 else 'FAIL'}] {label}: {n} {what.split(':')[0]}" + (":" +
what.split(":", 1)[1] if ":" in what else ""))
    if not ok:
        print("[note] a failed check means the lists below may be incomplete: fix the export or
explain the gap in KNOWN_GAPS, then run again")

    # ---- who each account is
    def person_of(platform, login, hint):
        if (platform, login) in ident:
            return ident[(platform, login)]
        if "@" in login:
            return login
        return hint if hint and "@" in hint else None

    n_in = {}
    for p, l, m, t in signins:
        n_in[(p, l)] = n_in.get((p, l), 0) + 1
    for a in accts:
        a["type"] = "service" if (a["platform"], a["login"]) in owners else a["type"]
        a["person"] = person_of(a["platform"], a["login"], a["hint"])
    by_login = {(a["platform"], a["login"]): a for a in accts}

    def type_of(p, l):
        if (p, l) in owners:
            return "service"
        return by_login[(p, l)]["type"] if (p, l) in by_login else "person"

    # ---- list 1: the user listing
    rows1 = []
    for a in sorted(accts, key=lambda a: (a["platform"], a["login"])):
        if not a["in_period"]:
            continue
        flags = []
        if a["type"] == "person":
            if a["person"] is None or a["person"] not in hr:
                flags.append("NO PERSON")
            elif a["person"] in leaver and a["enabled"]:
                flags.append("LEFT BUT ENABLED")
        else:
            o = owners.get((a["platform"], a["login"]), "")
            if not o or o not in hr or o in leaver:
                flags.append("SERVICE NO OWNER")
        rows1.append((a["platform"], a["login"], a["type"], a["person"] or "", "yes" if
a["enabled"] else "no",
                      n_in.get((a["platform"], a["login"]), 0), " + ".join(flags) if flags else "no
flag"))
    seen_logins = sorted({(p, l) for p, l, _, _ in signins if (p, l) not in by_login})
    for p, l in seen_logins:
        who_ = person_of(p, l, "")
        status = "LEFT, NO LONGER LISTED" if p != "snowflake" and who_ in leaver else "LOGIN NOT IN
LISTING"
        rows1.append((p, l, "(not in listing)", who_ or "", "", n_in[(p, l)], status))
    rows1.sort(key=lambda r: tuple(str(v) for v in r))
    population("soc2_1_user_listing", "User listing (accounts during the period)",
               "every account that existed at any time in the period (Snowflake: CREATED_ON to
DELETED_ON; Databricks and Redshift export only the "
               "accounts that exist at the export, so a removed account is not listed), plus any
login seen signing in that the listing lacks; "
```

```
                "one row per account", rows1, ["platform", "login", "account_type", "person",
"enabled_at_as_of", "signins_in_period", "status"], ok)
    show(rows1, lambda s: s != "no flag", 6)

    # ---- list 2: administrators
    rows2 = []
    for p, l, how, since in sorted(set(admin_rows)):
        a = by_login.get((p, l))
        rows2.append((p, l, (a or {}).get("person") or "", how, since, "approved" if (p, l) in
approved else "NOT ON ADMIN LIST"))
    rows2.sort(key=lambda r: tuple(str(v) for v in r))
    population("soc2_2_administrators", "Administrators at AS_OF against the approved list",
                "Snowflake: users holding a role named in ADMIN_ROLES (GRANTS_TO_USERS rows open at
the end of AS_OF); Redshift: superusers; "
                "Databricks: active members of " + ", ".join(DATABRICKS_ADMIN_GROUPS) + "; one row
per account and right", rows2,
                ["platform", "login", "person", "right", "since", "status"], ok)
    show(rows2, lambda s: s != "approved", 5)
    rows2b = sorted(admin_grants)
    population("soc2_2b_admin_grants_in_period", "Administrator role grants made in the period",
                "Snowflake GRANTS_TO_USERS rows for ADMIN_ROLES with CREATED_ON in the period; the
granter is a role, not a person; Databricks and "
                "Redshift: these exports do not record who made a grant", rows2b, ["granted_on",
"role", "grantee", "granted_by_role", "still_held"], ok)
    show(rows2b, lambda s: True, 4, 50)

    # ---- list 3: sign-ins against your policy
    groups = {}
    for p, l, m, t in signins:
        k = (p, l, m)
        g_ = groups.setdefault(k, [0, t, t])
        g_[0] += 1
        g_[1], g_[2] = min(g_[1], t), max(g_[2], t)
    rows3 = []
    for (p, l, m), (n, first, last) in sorted(groups.items()):
        typ = type_of(p, l)
        allowed = pol.get((p, typ))
        status = "NO POLICY SET" if allowed is None else ("within policy" if m.lower() in allowed
else "OUTSIDE POLICY")
        last_day = leaver.get(person_of(p, l, (by_login.get((p, l)) or {}).get("hint", "")))
        after = sum(1 for p_, l_, m_, t_ in signins if (p_, l_, m_) == (p, l, m) and last_day is
not None and t_.date() > last_day)
        detail = f"{after} after last day {last_day.isoformat()}" if after else ""
        if after:
            status = "AFTER LAST DAY" + ("" if status == "within policy" else " + " + status)
        rows3.append((p, l, typ, "sign-in", m, n, first.date().isoformat(),
last.date().isoformat(), detail, status))
    for who_, t, life in sorted(tokens, key=lambda x: (x[0], x[1], x[2] or -1)):
        status = "OUTSIDE POLICY" if life is None or life > max_token else "within policy"
        last_day = leaver.get(person_of("databricks", who_, ""))
        if last_day is not None and t.date() > last_day:
            status = "AFTER LAST DAY" + ("" if status == "within policy" else " + " + status)
        rows3.append(("databricks", who_, type_of("databricks", who_), "token created",
                      "generateDbToken", 1, t.date().isoformat(), t.date().isoformat(), "no expiry"
if life is None else f"lasts {life} days", status))
    rows3.sort(key=lambda r: tuple(str(v) for v in r))
    population("soc2_3_signins_against_policy", "Successful sign-ins by method, and tokens created,
against your policy",
```

```
                "successful sign-ins in the period (Snowflake IS_SUCCESS, Databricks status 200,
Redshift event " + REDSHIFT_SIGNIN_EVENT +
                "), one row per login and method; the allowed methods come from POLICY by account
type; AFTER LAST DAY counts sign-ins after the person's HR last day" +
                ("; a token is OUTSIDE POLICY with no expiry or a lifetime over " + str(max_token) +
" days" if "databricks" in exported else ""),
                rows3, ["platform", "login", "account_type", "what", "method", "count", "first_day",
"last_day", "detail", "status"], ok)
    show(rows3, lambda s: s != "within policy", 9)

    # ---- result
    f1 = sum(1 for r in rows1 if r[6] not in ("no flag", "LEFT, NO LONGER LISTED"))
    f2 = sum(1 for r in rows2 if r[5] != "approved")
    f3 = sum(1 for r in rows3 if r[9] != "within policy")
    print("\n== Result")
    print(f"input checks : {'all passed' if ok else 'FAILED, the lists above are not complete'}")
    print(f"not checked  : {', '.join(not_checked) if not_checked else 'none'}")
    print(f"follow-up    : {f1} account row(s) to explain, {f2} administrator row(s) not on the
approved list, {f3} sign-in or token row(s) outside policy or after a last day")
    if not ok or f1 + f2 + f3:
        verdict = "EXCEPTIONS FOUND"
    elif not_checked:
        verdict = "NOTHING TO FOLLOW UP ON THE PLATFORMS CHECKED; NOT CHECKED: " + ",
".join(not_checked)
    else:
        verdict = "NOTHING TO FOLLOW UP"
    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: clean generated records (python3 soc2_user_access_evidence.py --demo)

```
input     : GENERATED records (not a company's data), seed 230, plant=False
platforms : checked snowflake, databricks, redshift

== Input checks
[ok] HR file: rows with no email: 0 rows
[ok] HR file: last days not written as YYYY-MM-DD: 0 rows
[ok] login map: rows with no login or no email: 0 rows
[ok] USERS: empty export: 0 export
[ok] USERS: rows with no NAME: 0 rows
[ok] USERS: CREATED_ON not written as YYYY-MM-DD HH:MM:SS: 0 rows
[ok] USERS: DELETED_ON or LAST_SUCCESS_LOGIN not written as YYYY-MM-DD HH:MM:SS: 0 rows
[ok] USERS: DISABLED is not true or false: 0 rows
[ok] LOGIN_HISTORY: EVENT_TIMESTAMP or IS_SUCCESS unreadable, or a successful sign-in with no
USER_NAME: 0 rows
[ok] GRANTS_TO_USERS: dates unreadable: 0 rows
[ok] ADMIN_ROLES: a name that matches no grant in GRANTS_TO_USERS (a typo hides every holder): 0
names
```

```
[ok] LOGIN_HISTORY: a user's LAST_SUCCESS_LOGIN day has no successful sign-in on it (filtered or
short export): 0 users
[ok] SCIM users: empty export: 0 export
[ok] SCIM users: userName empty or listed twice: 0 rows
[ok] SCIM users: active is not true or false: 0 rows
[ok] system.access.audit: event_time not written as YYYY-MM-DD HH:MM:SS: 0 rows
[ok] system.access.audit: user_identity that is not JSON: 0 rows
[ok] system.access.audit: accounts actions that look like a sign-in but are not in
DATABRICKS_LOGIN_ACTIONS: 0 actions
[ok] system.access.audit: sign-in rows with no readable user or request_params: 0 rows
[ok] system.access.audit: sign-in or token rows with no readable status: 0 rows
[ok] system.access.audit: token rows with no creator or an expiry that is not a number: 0 rows
[ok] SVV_USER_INFO: empty export: 0 export
[ok] SVV_USER_INFO: user_name empty or listed twice: 0 rows
[ok] SVV_USER_INFO: superuser is not true or false: 0 rows
[ok] SYS_CONNECTION_LOG: empty export (a non-superuser sees no rows): 0 export
[ok] SYS_CONNECTION_LOG: record_time not written as YYYY-MM-DD HH:MM:SS, or no user on a sign-in: 0
rows
[ok] snowflake: days with no sign-in record from 2026-07-01 to 2026-10-15: 0 days
[ok] databricks: days with no sign-in record from 2026-07-01 to 2026-10-15: 0 days
[ok] redshift: days with no sign-in record from 2026-07-01 to 2026-10-15: 0 days

== User listing (accounts during the period): soc2_1_user_listing
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : every account that existed at any time in the period (Snowflake: CREATED_ON to
DELETED_ON; Databricks and Redshift export only the accounts that exist at the export, so a removed
account is not listed), plus any login seen signing in that the listing lacks; one row per account
run at    : 2026-11-01 12:06:04 UTC
rows      : 18
sha256    : 0ebe371093250d154f626069812b51e034b6b9959fb877cdf0241af3738e99ec
  redshift | eli | (not in listing) | p06@example.test | - | 8 | LEFT, NO LONGER LISTED

== Administrators at AS_OF against the approved list: soc2_2_administrators
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : Snowflake: users holding a role named in ADMIN_ROLES (GRANTS_TO_USERS rows open at the
end of AS_OF); Redshift: superusers; Databricks: active members of admins; one row per account and
right
run at    : 2026-11-01 12:06:04 UTC
rows      : 4
sha256    : c8f436efb4bf9bc2c3f2f6cee2c29d0f7c6cedf1ee90ebec3e3316c6a488046b
  databricks | p01@example.test | p01@example.test | group admins | - | approved
  redshift | ana | p01@example.test | superuser | - | approved
  snowflake | p01 | p01@example.test | role ACCOUNTADMIN | 2026-01-05 | approved
  snowflake | p02 | p02@example.test | role SECURITYADMIN | 2026-08-12 | approved

== Administrator role grants made in the period: soc2_2b_admin_grants_in_period
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : Snowflake GRANTS_TO_USERS rows for ADMIN_ROLES with CREATED_ON in the period; the
granter is a role, not a person; Databricks and Redshift: these exports do not record who made a
grant
run at    : 2026-11-01 12:06:04 UTC
rows      : 1
sha256    : 63daf5720760ea655da872b577d48ae3a5b24face2bfeeda7dc43bdddb2d34c4
  2026-08-12 | SECURITYADMIN | p02 | ACCOUNTADMIN | yes

== Successful sign-ins by method, and tokens created, against your policy:
soc2_3_signins_against_policy
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
```

```
filters   : successful sign-ins in the period (Snowflake IS_SUCCESS, Databricks status 200,
Redshift event authenticated), one row per login and method; the allowed methods come from POLICY
by account type; AFTER LAST DAY counts sign-ins after the person's HR last day; a token is OUTSIDE
POLICY with no expiry or a lifetime over 90 days
run at    : 2026-11-01 12:06:04 UTC
rows      : 20
sha256    : 5d6a29f31063c94126c57fe4a8eb4993a9d797c6ba5f5e4bb9b85c8387c86402

== Result
input checks : all passed
not checked  : none
follow-up    : 0 account row(s) to explain, 0 administrator row(s) not on the approved list, 0
sign-in or token row(s) outside policy or after a last day
RESULT: NOTHING TO FOLLOW UP
```

Run 2: generated records with planted faults (python3 soc2_user_access_evidence.py --demo --plant)

```
input     : GENERATED records (not a company's data), seed 230, plant=True
platforms : checked snowflake, databricks, redshift

== Input checks
[ok] HR file: rows with no email: 0 rows
[FAIL] HR file: last days not written as YYYY-MM-DD: 1 rows
[ok] login map: rows with no login or no email: 0 rows
[ok] USERS: empty export: 0 export
[ok] USERS: rows with no NAME: 0 rows
[ok] USERS: CREATED_ON not written as YYYY-MM-DD HH:MM:SS: 0 rows
[ok] USERS: DELETED_ON or LAST_SUCCESS_LOGIN not written as YYYY-MM-DD HH:MM:SS: 0 rows
[ok] USERS: DISABLED is not true or false: 0 rows
[ok] LOGIN_HISTORY: EVENT_TIMESTAMP or IS_SUCCESS unreadable, or a successful sign-in with no
USER_NAME: 0 rows
[ok] GRANTS_TO_USERS: dates unreadable: 0 rows
[ok] ADMIN_ROLES: a name that matches no grant in GRANTS_TO_USERS (a typo hides every holder): 0
names
[FAIL] LOGIN_HISTORY: a user's LAST_SUCCESS_LOGIN day has no successful sign-in on it (filtered or
short export): 1 users
[ok] SCIM users: empty export: 0 export
[ok] SCIM users: userName empty or listed twice: 0 rows
[ok] SCIM users: active is not true or false: 0 rows
[ok] system.access.audit: event_time not written as YYYY-MM-DD HH:MM:SS: 0 rows
[ok] system.access.audit: user_identity that is not JSON: 0 rows
[ok] system.access.audit: accounts actions that look like a sign-in but are not in
DATABRICKS_LOGIN_ACTIONS: 0 actions
[ok] system.access.audit: sign-in rows with no readable user or request_params: 0 rows
[ok] system.access.audit: sign-in or token rows with no readable status: 0 rows
[ok] system.access.audit: token rows with no creator or an expiry that is not a number: 0 rows
[ok] SVV_USER_INFO: empty export: 0 export
[ok] SVV_USER_INFO: user_name empty or listed twice: 0 rows
[ok] SVV_USER_INFO: superuser is not true or false: 0 rows
[ok] SYS_CONNECTION_LOG: empty export (a non-superuser sees no rows): 0 export
[ok] SYS_CONNECTION_LOG: record_time not written as YYYY-MM-DD HH:MM:SS, or no user on a sign-in: 0
rows
[ok] snowflake: days with no sign-in record from 2026-07-01 to 2026-10-15: 0 days
[ok] databricks: days with no sign-in record from 2026-07-01 to 2026-10-15: 0 days
[FAIL] redshift: days with no sign-in record from 2026-07-01 to 2026-10-15: 1 days: 2026-08-19
```

```
[note] a failed check means the lists below may be incomplete: fix the export or explain the gap in
KNOWN_GAPS, then run again

== User listing (accounts during the period): soc2_1_user_listing   ** INCOMPLETE: see input checks
**
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : every account that existed at any time in the period (Snowflake: CREATED_ON to
DELETED_ON; Databricks and Redshift export only the accounts that exist at the export, so a removed
account is not listed), plus any login seen signing in that the listing lacks; one row per account
run at    : 2026-11-01 12:06:08 UTC
rows      : 21
sha256    : fc0298281701fad612427135132afea5c0fd45ab370eed93dda12ec64ebabe26
  databricks | p06@example.test | person | p06@example.test | yes | 7 | LEFT BUT ENABLED
  databricks | temp.consultant@example.test | person | temp.consultant@example.test | yes | 0 | NO
PERSON
  redshift | eli | person | p06@example.test | yes | 9 | LEFT BUT ENABLED
  redshift | etl_loader | service | - | yes | 91 | SERVICE NO OWNER
  snowflake | contractor1 | person | - | yes | 1 | NO PERSON
  snowflake | p06 | person | p06@example.test | yes | 8 | LEFT BUT ENABLED
  snowflake | sf_ghost | (not in listing) | - | - | 7 | LOGIN NOT IN LISTING

== Administrators at AS_OF against the approved list: soc2_2_administrators   ** INCOMPLETE: see
input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : Snowflake: users holding a role named in ADMIN_ROLES (GRANTS_TO_USERS rows open at the
end of AS_OF); Redshift: superusers; Databricks: active members of admins; one row per account and
right
run at    : 2026-11-01 12:06:08 UTC
rows      : 7
sha256    : b310e10ee529726a434def3d8d768585217325055d6e117db7c967cb7cb1220a
  databricks | p01@example.test | p01@example.test | group admins | - | approved
  databricks | p03@example.test | p03@example.test | group admins | - | NOT ON ADMIN LIST
  redshift | ana | p01@example.test | superuser | - | approved
  redshift | dee | p04@example.test | superuser | - | NOT ON ADMIN LIST
  snowflake | p01 | p01@example.test | role ACCOUNTADMIN | 2026-01-05 | approved
  snowflake | p02 | p02@example.test | role SECURITYADMIN | 2026-08-12 | approved
  snowflake | p03 | p03@example.test | role ACCOUNTADMIN | 2026-09-02 | NOT ON ADMIN LIST

== Administrator role grants made in the period: soc2_2b_admin_grants_in_period   ** INCOMPLETE:
see input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : Snowflake GRANTS_TO_USERS rows for ADMIN_ROLES with CREATED_ON in the period; the
granter is a role, not a person; Databricks and Redshift: these exports do not record who made a
grant
run at    : 2026-11-01 12:06:08 UTC
rows      : 2
sha256    : e33edd2026eba487ec7d65341eff6cebf2e3f1f7c4c2f30b03b0ff7f285aaade
  2026-08-12 | SECURITYADMIN | p02 | ACCOUNTADMIN | yes
  2026-09-02 | ACCOUNTADMIN | p03 | ACCOUNTADMIN | yes

== Successful sign-ins by method, and tokens created, against your policy:
soc2_3_signins_against_policy   ** INCOMPLETE: see input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); accounts and administrators as of 2026-10-15
filters   : successful sign-ins in the period (Snowflake IS_SUCCESS, Databricks status 200,
Redshift event authenticated), one row per login and method; the allowed methods come from POLICY
by account type; AFTER LAST DAY counts sign-ins after the person's HR last day; a token is OUTSIDE
POLICY with no expiry or a lifetime over 90 days
run at    : 2026-11-01 12:06:08 UTC
```

```
rows      : 27
sha256    : 089b0388a5b678c1938e953b58e961ed76bbb2d9e289222873cc6a90cf2ae016
  databricks | p02@example.test | person | token created | generateDbToken | 1 | 2026-09-15 | 2026-
09-15 | no expiry | OUTSIDE POLICY
  databricks | p04@example.test | person | sign-in | tokenLogin | 1 | 2026-09-25 | 2026-09-25 | - |
OUTSIDE POLICY
  databricks | svc-jobs@example.test | service | token created | generateDbToken | 1 | 2026-09-20 |
2026-09-20 | lasts 365 days | OUTSIDE POLICY
  redshift | cy | person | sign-in | password | 6 | 2026-09-01 | 2026-09-26 | - | OUTSIDE POLICY
  redshift | eli | person | sign-in | iam | 9 | 2026-07-01 | 2026-08-20 | 1 after last day 2026-08-
07 | AFTER LAST DAY
  snowflake | p03 | person | sign-in | PASSWORD | 4 | 2026-09-11 | 2026-09-26 | - | OUTSIDE POLICY

== Result
input checks : FAILED, the lists above are not complete
not checked  : none
follow-up    : 7 account row(s) to explain, 3 administrator row(s) not on the approved list, 6
sign-in or token row(s) outside policy or after a last day
RESULT: EXCEPTIONS FOUND
```

How to read the results

- **INCOMPLETE** on a list means an input check failed: the list may be missing rows. Fix the export, or explain a quiet day in `KNOWN_GAPS`, before you show it to anyone.
- **NO PERSON** is an account HR cannot match. It may be a contractor or a missing login-map row; the list cannot tell which.
- **LEFT BUT ENABLED** is an HR leaver whose account is still enabled. **SERVICE NO OWNER** is a service account nobody answers for.
- **LOGIN NOT IN LISTING** is a login that signed in but the listing lacks. On Snowflake the likely cause is a filtered export; on Databricks and Redshift it can also be an account deleted since; find out which before you show the listing as complete.
- **LEFT, NO LONGER LISTED** is a Databricks or Redshift login that HR says has left and the current user list no longer shows. It is informational and is not counted.
- **NOT ON ADMIN LIST** is an administrator your approved list does not name. The list may be out of date, so it is a question, not a verdict.
- **OUTSIDE POLICY** is a sign-in method or a token your own policy does not allow for that account type. The script compares strings with your policy; whether the policy is right is yours to decide.
- **AFTER LAST DAY** is a sign-in or token after the person's HR last day. It is counted whether or not the account is still listed.
- Keep each run's settings, its SHA-256 lines and the CSVs it wrote. Your auditor decides what they show.

Method

- **Input:** generated records written by the script as CSV in each platform's published column names (the links in step 1), with a fixed seed (230). The HR file, login map, owner list and approved list are generated too. The people, logins and accounts are invented.
- **Period:** 2026-07-01 to 2026-09-30, UTC, inclusive, for the sign-ins and the accounts. Administrators and enabled accounts are read as of the end of `AS_OF` (2026-10-15). Completeness is checked for every day from the period's start to `AS_OF` on each platform.
- **A successful sign-in** is a Snowflake LOGIN_HISTORY row with IS_SUCCESS yes, a Databricks `accounts` audit row whose action is in `DATABRICKS_LOGIN_ACTIONS` with a status of 200, or a Redshift SYS_CONNECTION_LOG row with the event `authenticated`. A refused attempt is not counted.
- **Who is a service account:** any login in your owner file first, then a Snowflake user whose type is not blank or PERSON. Everything else is a person, in every list.
- **The demo policy** is an example, not a recommendation of what any account type should use: people sign in by single sign-on, IAM or a password with a second factor; service accounts by key pair, token, IAM or (on Redshift) a password; and tokens last at most ninety days.
- **Unreadable input is counted, not dropped:** a failed input check marks every list INCOMPLETE. The Limits below list what the script cannot see.
- **Input checks in the runs:** of the twenty-nine input checks, five failed in the runs shown; the other twenty-four are written to fail on the fault their labels name and were not triggered.

Run it on your own data

To run it on your own data, change only the settings block at the top of the download. Set EXPORT_DIR to the folder holding your exports. Set PERIOD_START, PERIOD_END and AS_OF. Set PLATFORMS_EXPORTED to the platforms you exported. Set ADMIN_ROLES to your Snowflake administrator roles, write your sign-in policy into POLICY, and set MAX_TOKEN_DAYS if you export Databricks. If your HR file's headings are not email and last_day, set HR_EMAIL_COL and HR_LAST_DAY_COL. If a file has another name, change its entry in FILES. Export each platform file with the platform's own column names and timestamps as YYYY-MM-DD HH:MM:SS in UTC, and keep the headings the script names in the login map, service owner and approved administrator files.

For each login that is not an email address and has no email in the platform's own record, add a row to the login map. Change nothing else, and run it without --demo.

Checklist

Done	Step
☐	Ask your auditor for their sample request and keep it with the run.
☐	Export every file in the table above, as a user who can see every row (a Redshift superuser for SVV_USER_INFO and SYS_CONNECTION_LOG).
☐	Set the timestamps in the exports to UTC and YYYY-MM-DD HH:MM:SS.
☐	Set ADMIN_ROLES, POLICY and MAX_TOKEN_DAYS to your own choices; they have no default.
☐	List your own sign-in method strings first (run once, read the sign-in list) before you write the policy.
☐	Add every login without an email to the login map; list every service account with an owner.
☐	Run the script and read the input checks first; do not show a list marked INCOMPLETE.
☐	Explain each NO PERSON, LEFT BUT ENABLED and SERVICE NO OWNER row.
☐	Show the LOGIN NOT IN LISTING result with the user listing.
☐	Take each NOT ON ADMIN LIST row to whoever keeps the approved list.
☐	Take each OUTSIDE POLICY row to the owner of the policy.
☐	Explain each AFTER LAST DAY row: a sign-in or token after the person's HR last day.
☐	Keep the settings, the SHA-256 lines and the CSVs of each run.

Limits

- **This is not a client result.** The records are generated. The runs show that the script finds planted faults. They say nothing about any real company or how often such faults occur.
- **We have not run it on a real account.** Real exports can differ from the documentation. These are our assumptions, so open one of your own rows and check: Snowflake's LOGIN_HISTORY `USER_NAME` is the user's `NAME` in USERS; the Databricks sign-in event carries the user in `request_params` under `user`; `tokenExpirationTime` is Unix time in milliseconds, with an empty or zero value meaning no expiry (the Databricks page names the parameter but gives neither its format nor a default token lifetime; Databricks: audit log reference (read 2026-11-01)); a successful Databricks sign-in has `statusCode` 200 in `response`; a non-superuser's export of SYS_CONNECTION_LOG comes back empty; the user lists of Databricks and Redshift hold only accounts that still exist; and a blank or PERSON Snowflake type is a person, any other a service account.

- **The criteria are not a checklist.** We describe CC6.1 to CC6.3 in our own words and do not quote them. Which lists an auditor asks for, and in what form, is theirs to say.
- **The listing says who had an account, not whether the access was right.** It shows no approvals and no access reviews. We read the Databricks and Redshift lists as of the export day, so a removal is seen only through a login in the sign-in records that HR matches to a leaver. On Redshift, SVV_USER_INFO has no disabled flag, so every listed account counts as enabled. Redshift's `external_user_id` is the identity provider's identifier (Amazon Redshift: SVV_USER_INFO (read 2026-11-01)); we use it as an email only when it is one, so add Redshift logins to the login map.
- **Administrators are only those the settings name.** A Snowflake role granted to another role, a nested Databricks group, a Databricks account admin (a role, not group membership) and a Redshift role other than superuser are not followed. Databricks service principals are not in the users export at all.
- **Admin grants with a date and a granter exist only for Snowflake**, and the granter is a role. The Databricks and Redshift exports here do not record who made a grant.
- **Tokens:** only tokens created in the period are listed. A token created earlier and still valid is not, and a revoked token is not followed.
- **Sign-in methods are strings.** A Databricks method is the audit log's action name, and its `authenticationMethod` parameter is not read. A Snowflake second factor that is empty means none was recorded.
- **The empty-day check can fire on a quiet day,** and cannot see a day that has some records but not all. On Databricks only `accounts` rows count for a day, so make sure your audit export includes that service.
- **Timestamps with no zone suffix are taken as UTC.** Check the time zone of your own export session and set it to UTC before you export.

When this is not enough

- **You need more than one day's pull.** The script reads exports taken on one day. Keep each pull's settings and the lists as produced, and take the exports often enough that your platforms' own retention never leaves a gap.
- **Your platforms are not these three,** or you cannot export the records named in step 1.
- **The criteria are more than three lists.** Approvals, reviews and the policy itself sit in your own systems, not in the platforms' records.
- **Audit advice is not here.** Whether a list is enough is for your auditor.

Sources

Document	Read
AICPA: 2017 Trust Services Criteria (With Revised Points of Focus, 2022), CC6.1 to CC6.3; described, not quoted	2026-11-01
Snowflake: USERS	2026-11-01
Snowflake: LOGIN_HISTORY	2026-11-01
Snowflake: GRANTS_TO_USERS	2026-11-01
Databricks: list users (SCIM)	2026-11-01
Databricks: audit log system table	2026-11-01
Databricks: audit log reference	2026-11-01
Azure Databricks: audit log reference	2026-11-01
Amazon Redshift: SVV_USER_INFO	2026-11-01
Amazon Redshift: SYS_CONNECTION_LOG	2026-11-01
Data Control Tower (solution page)	2026-11-01

See leavers and access changes every day

Data Control Tower deploys into your own Azure subscription from a template your IT team can read before it runs. Every day it reads the records your data platforms already keep, and it never writes to them. It lists every access change and who made it, including access someone gave by hand, and checks leavers against your HR file, across every platform you connect.

See what Data Control Tower checks