# ISO/IEC 27001:2022 Evidence Mapper (DB-driven, Read-only)

Goal: Produce auditor-friendly evidence views from the waf_verification PostgreSQL database and map them to ISO/IEC 27001:2022 control intents, specifically: • A.12.6.1 — Management of technical vulnerabilities (detection, monitoring, remediation evidence) • A.16.1.7 — Collection of evidence (traceable logs, timestamps, integrity of evidence trail) • A.18.2.3 — Technical compliance review (repeatable compliance checks and outputs)

# Non-negotiables (for audit reliability):

1. No enforcement actions in this notebook (no ipset/iptables/nftables changes).
2. No secrets stored in notebook cells (DB credentials must come from /home/oba/AstroMap/.waf_db.env).
3. No schema guessing. Every query must match actual DB columns. If a column is missing, we stop and correct.
4. Verification-first: We run and validate each cell before proceeding.
5. HOME_IP safety: exclude the protected home IP from any candidate/summary tables.

In [1]:
```
# Cell 1

import os
import pandas as pd
from datetime import datetime, timedelta, timezone
from sqlalchemy import create_engine, text

# ---- Required external env file (cron-safe) ----
ENV_FILE = "/home/oba/AstroMap/.waf_db.env"

def load_env_file(path: str) -> None:
    if not os.path.exists(path):
        raise FileNotFoundError(
            f"Cell 1: Missing env file: {path}\n"
            "Create it with DB_HOST, DB_PORT, DB_NAME, DB_USER, DB_PASS (anc
        )
    for raw in open(path, "r", encoding="utf-8"):
        line = raw.strip()
        if not line or line.startswith("#") or "=" not in line:
            continue
        k, v = line.split("=", 1)
        k = k.strip()
        v = v.strip().strip('"').strip("'")
        # Do not overwrite values already set in the process environment
        os.environ.setdefault(k, v)
```

```
load_env_file(ENV_FILE)

# ---- Inputs (no hardcoded secrets) ----
DB_HOST = os.environ.get("DB_HOST", "")
DB_PORT = os.environ.get("DB_PORT", "5432")
DB_NAME = os.environ.get("DB_NAME", "")
DB_USER = os.environ.get("DB_USER", "")
DB_PASS = os.environ.get("DB_PASS", "")

HOME_IP = os.environ.get("HOME_IP", "66.241.78.7")
WINDOW_HOURS = int(os.environ.get("WINDOW_HOURS", "72"))  # initial ISO winc

missing = [k for k, v in {
    "DB_HOST": DB_HOST,
    "DB_NAME": DB_NAME,
    "DB_USER": DB_USER,
    "DB_PASS": DB_PASS,
}.items() if not v]

if missing:
    raise RuntimeError(f"Cell 1: Missing required env vars: {missing} (check

# ---- DB engine ----
url = f"postgresql+psycopg2://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NA
engine = create_engine(url, pool_pre_ping=True)

# ---- Time window ----
end_utc = datetime.now(timezone.utc)
start_utc = end_utc - timedelta(hours=WINDOW_HOURS)

print(f"[Cell 1] DB={DB_NAME} Host={DB_HOST}:{DB_PORT}")
print(f"[Cell 1] Window UTC: {start_utc.isoformat()} -> {end_utc.isoformat()
print(f"[Cell 1] HOME_IP excluded: {HOME_IP}")
```

```
[Cell 1] DB=waf_verification Host=127.0.0.1:5432
[Cell 1] Window UTC: 2026-02-26T21:45:03.205453+00:00 -> 2026-03-01T21:45:0
3.205453+00:00
[Cell 1] HOME_IP excluded: 66.241.78.7
```

## Cell 2 — Build the canonical evidence dataset (minimal columns, no guesswork)

Purpose: Pull a strict, schema-safe dataset from waf.events for the selected time window. This dataset will be the single source of truth for ISO evidence tables/figures downstream.

Success criteria:

- Query executes with no missing-column errors.
- Output includes:
- row count
- unique IP count (excluding HOME_IP)
- verdict distribution

In [2]:
```python
# Cell 3 — Python (Canonical SELECT from waf.events)

# Canonical SELECT: only columns we have already proven exist in your enviro
# (ts_utc, ip, verdict, expected_action, source_log)

sql = text("""
SELECT
  ts_utc,
  ip,
  verdict,
  expected_action,
  source_log
FROM waf.events
WHERE ts_utc >= :start
  AND ts_utc <  :end
""")

df_events = pd.read_sql(sql, engine, params={"start": start_utc, "end": end_

# Normalize IP (strip /32 etc.)
df_events["ip_norm"] = df_events["ip"].astype(str).str.split("/").str[0].str

# Safety exclusion
df_events = df_events[df_events["ip_norm"] != HOME_IP].copy()

print(f"[Cell 3] rows={len(df_events)} unique_ips={df_events['ip_norm'].nuni
print("[Cell 3] verdict counts:")
print(df_events["verdict"].value_counts(dropna=False))

df_events.head(10)
```

```
[Cell 3] rows=12293 unique_ips=426
[Cell 3] verdict counts:
verdict
SUSPICIOUS    8654
BENIGN        3149
TRACE          490
Name: count, dtype: int64
```

Out[2]:

| | ts_utc | ip | verdict | expected_action | source_log | ip_nor |
|---|---|---|---|---|---|---|
| **0** | 2026-02-26 21:56:08+00:00 | 43.159.128.247 | TRACE | MONITOR | detections.log | 43.159.128.24 |
| **1** | 2026-02-26 21:59:43+00:00 | 206.168.34.204 | TRACE | MONITOR | detections.log | 206.168.34.20 |
| **2** | 2026-02-26 22:01:38+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 127.0.0 |
| **3** | 2026-02-26 22:01:39+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 127.0.0 |
| **4** | 2026-02-26 22:14:24+00:00 | 43.132.214.228 | TRACE | MONITOR | detections.log | 43.132.214.22 |
| **5** | 2026-02-26 22:14:26+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 127.0.0 |
| **6** | 2026-02-26 22:15:06+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 127.0.0 |
| **7** | 2026-02-26 22:15:08+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 127.0.0 |
| **8** | 2026-02-26 22:17:43+00:00 | 49.51.50.147 | TRACE | MONITOR | detections.log | 49.51.50.14 |
| **9** | 2026-02-26 22:17:44+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 127.0.0 |

# Cell 4 — ISO Control Mapping: Evidence Preparation (Read-Only)

**Purpose (Auditor-facing):**
This cell prepares a **canonical, read-only evidence table** from the validated event dataset (Cell 3) and explicitly maps observable fields to ISO/IEC 27001:2022 control intent.

**Controls addressed (foundation):**

- **A.12.6.1** — Identification and monitoring of technical vulnerabilities (observable hostile activity).
- **A.16.1.7** — Collection and preservation of evidence (timestamps, immutable logs).
- **A.18.2.3** — Technical compliance review (repeatable, queryable datasets).

**Important constraints:**

- No enforcement logic.

- No interpretation or scoring.
- No data mutation.
- HOME_IP already excluded upstream.

This table is the **single source of truth** for all subsequent ISO evidence views.

In [3]:
```
# Cell 4.8 — Database Schema Discovery (Read-Only, No Assumptions)

import os
import sys
import pandas as pd
from sqlalchemy import create_engine, text

# ---- 1) Runtime identity (helps detect wrong kernel/env)
print("[Cell 4.8] python:", sys.executable)
print("[Cell 4.8] python_version:", sys.version.split()[0])
print("[Cell 4.8] pandas:", pd.__version__)

# ---- 2) Required DB env vars (must be present under nbconvert too)
DB_HOST = os.getenv("DB_HOST", "")
DB_PORT = os.getenv("DB_PORT", "5432")
DB_NAME = os.getenv("DB_NAME", "")
DB_USER = os.getenv("DB_USER", "")
DB_PASS = os.getenv("DB_PASS", "")

missing = [k for k,v in [("DB_HOST",DB_HOST),("DB_NAME",DB_NAME),("DB_USER",
if missing:
    raise RuntimeError(
        f"Cell 4.8: Missing DB env vars in notebook runtime: {missing}\n"
        "nbconvert/cron runs must pass these vars into the kernel. Verify yo
    )

print(f"[Cell 4.8] DB target: {DB_USER}@{DB_HOST}:{DB_PORT}/{DB_NAME}")

# ---- 3) Connect
dsn = f"postgresql+psycopg2://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NA
engine = create_engine(dsn, pool_pre_ping=True)

# ---- 4) Introspect schema: find relations that contain expected ISO evider
q = text("""
SELECT
  c.table_schema,
  c.table_name,
  c.column_name,
  c.data_type
FROM information_schema.columns c
WHERE c.column_name IN ('ts_utc','ip_norm','verdict','expected_action','sour
  AND c.table_schema NOT IN ('pg_catalog','information_schema')
ORDER BY c.table_schema, c.table_name, c.column_name;
""")

with engine.connect() as conn:
    rows = conn.execute(q).mappings().all()

if not rows:
```

```
    raise RuntimeError(
        "Cell 4.8: No relations found with any of the expected columns "
        "(ts_utc/ip_norm/verdict/expected_action/source_log).\n"
        "This usually means: wrong DB, wrong user/privileges, or schema char
    )

df_cols = pd.DataFrame(rows)

print("\n[Cell 4.8] Relations/columns found (sample):")
display(df_cols.head(50))

# ---- 5) Candidates with ts_utc (strong signal)
cand = (
    df_cols[df_cols["column_name"] == "ts_utc"]
    .groupby(["table_schema","table_name"])
    .size()
    .reset_index(name="has_ts_utc")
    .sort_values(["table_schema","table_name"])
)

print("\n[Cell 4.8] Candidate relations containing ts_utc (use schema-qualif
display(cand)

print("\n[Cell 4.8] TIP: pick one and set e.g. SOURCE_RELATION='schema.table
```

```
[Cell 4.8] python: /home/oba/miniconda3/envs/tfA/bin/python
[Cell 4.8] python_version: 3.10.18
[Cell 4.8] pandas: 2.3.3
[Cell 4.8] DB target: waf_reader@127.0.0.1:5432/waf_verification

[Cell 4.8] Relations/columns found (sample):
```

| | column_name | data_type | table_name | table_schema |
|---|---|---|---|---|
| **0** | source_log | text | events | nginxdata |
| **1** | ts_utc | timestamp with time zone | events | nginxdata |
| **2** | expected_action | text | events | waf |
| **3** | source_log | text | events | waf |
| **4** | ts_utc | timestamp with time zone | events | waf |
| **5** | verdict | text | events | waf |
| **6** | expected_action | text | events_compat | waf |
| **7** | source_log | text | events_compat | waf |
| **8** | verdict | text | events_compat | waf |
| **9** | source_log | text | host_auth_events | waf |
| **10** | ts_utc | timestamp with time zone | host_auth_events | waf |
| **11** | source_log | text | log_offsets | waf |
| **12** | source_log | text | mail_auth_events | waf |
| **13** | ts_utc | timestamp with time zone | mail_auth_events | waf |
| **14** | source_log | text | runs | waf |

```
[Cell 4.8] Candidate relations containing ts_utc (use schema-qualified nam
e):
```

| | table_schema | table_name | has_ts_utc |
|---|---|---|---|
| **0** | nginxdata | events | 1 |
| **1** | waf | events | 1 |
| **2** | waf | host_auth_events | 1 |
| **3** | waf | mail_auth_events | 1 |

```
[Cell 4.8] TIP: pick one and set e.g. SOURCE_RELATION='schema.table' in Cell
4.9.
```

# Cell 4.9 — DB Sanity Check (Data Availability & Window Validation)

This cell validates that:

- DB connection parameters are present in the notebook runtime.
- The source table/view contains data at all.
- The selected audit window overlaps actual data (min/max ts_utc).

- The row count in-window is non-zero before building df_events.

If this cell reports 0 rows in-window, downstream ISO evidence cells must fail by design.

In [4]:

```
import os
import pandas as pd
from sqlalchemy import create_engine, text

# ============================================================
# Cell 4.9 — DB Sanity + Window Alignment (EXECUTION-SAFE)
#
# Goal:
# - Ensure DB env vars exist
# - Connect read-only
# - Determine the correct SOURCE_RELATION (allow override)
# - Compute a window that actually overlaps available data
#   (auto-snaps to MAX(ts_utc) if the default window is empty)
# - Export globals: SOURCE_RELATION, audit_start_utc, audit_end_utc, engine
# ============================================================

# ---- 1) Required DB env vars (must be present under nbconvert too)
DB_HOST = os.getenv("DB_HOST", "")
DB_PORT = os.getenv("DB_PORT", "5432")
DB_NAME = os.getenv("DB_NAME", "")
DB_USER = os.getenv("DB_USER", "")
DB_PASS = os.getenv("DB_PASS", "")

missing = [k for k, v in [("DB_HOST", DB_HOST), ("DB_NAME", DB_NAME), ("DB_U
if missing:
    raise RuntimeError(
        f"[Cell 4.9] Missing DB env vars in notebook runtime: {missing}\n"
        "Cron/nbconvert must source /home/oba/AstroMap/.waf_db.env before ex
    )

print(f"[Cell 4.9] DB target: {DB_USER}@{DB_HOST}:{DB_PORT}/{DB_NAME}")

# ---- 2) Desired window (default: last 48h in UTC)
DEFAULT_WINDOW_HOURS = int(os.getenv("ISO_DEFAULT_WINDOW_HOURS", "48"))

# IMPORTANT: pandas 2.3+ makes Timestamp.utcnow() tz-aware.
# Use now(tz="UTC") which is unambiguous and safe.
desired_end_utc = pd.Timestamp.now(tz="UTC")
desired_start_utc = desired_end_utc - pd.Timedelta(hours=DEFAULT_WINDOW_HOUR

# Allow explicit override from earlier cells if you already define these glo
if "audit_start_utc" in globals() and "audit_end_utc" in globals():
    try:
        desired_start_utc = pd.Timestamp(globals()["audit_start_utc"]).tz_co
        desired_end_utc   = pd.Timestamp(globals()["audit_end_utc"]).tz_conv
    except Exception:
        # If upstream globals are malformed, fall back to default without fa
        pass

print(f"[Cell 4.9] Desired audit window UTC: {desired_start_utc.isoformat()}
```

```python
# ---- 3) Choose your source relation here (override allowed)
# Prefer explicit env var if you have it already set correctly.
SOURCE_RELATION = os.getenv("ISO_SOURCE_RELATION", "").strip()

dsn = f"postgresql+psycopg2://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NA
engine = create_engine(dsn, pool_pre_ping=True)

def _list_candidate_relations(conn):
    """
    Return relations that have a 'ts_utc' column (best-effort).
    This is read-only and prevents hardcoding a wrong table name.
    """
    q = text("""
        SELECT table_schema, table_name
        FROM information_schema.columns
        WHERE column_name = 'ts_utc'
          AND table_schema NOT IN ('pg_catalog','information_schema')
        GROUP BY table_schema, table_name
        ORDER BY table_schema, table_name
    """)
    return [(r["table_schema"], r["table_name"]) for r in conn.execute(q).ma

def _has_columns(conn, schema, table, cols):
    q = text("""
        SELECT column_name
        FROM information_schema.columns
        WHERE table_schema = :s AND table_name = :t
    """)
    have = {r["column_name"] for r in conn.execute(q, {"s": schema, "t": tab
    return all(c in have for c in cols)

def _pick_best_relation(conn, candidates):
    """
    Score candidates based on presence of expected columns.
    We do NOT change DB. We just pick the correct existing relation.
    """
    want = ["ts_utc", "verdict", "ip_norm", "source_log"]
    scored = []
    for schema, table in candidates:
        score = 0
        for c in want:
            if _has_columns(conn, schema, table, [c]):
                score += 5
        # slight preference for a 'waf' schema if present (your output showe
        if schema == "waf":
            score += 1
        scored.append((score, schema, table))
    scored.sort(reverse=True)
    return scored

def _count_rows(conn, rel):
    q = text(f"SELECT COUNT(*) AS n FROM {rel}")
    return int(conn.execute(q).scalar() or 0)

def _minmax_ts(conn, rel):
```

```
    q = text(f"SELECT MIN(ts_utc) AS min_ts, MAX(ts_utc) AS max_ts FROM {rel
    row = conn.execute(q).mappings().first()
    return row["min_ts"], row["max_ts"]

def _count_in_window(conn, rel, since_utc, until_utc):
    q = text(f"""
        SELECT COUNT(*) AS n
        FROM {rel}
        WHERE ts_utc >= :since AND ts_utc < :until
    """)
    return int(conn.execute(q, {"since": since_utc, "until": until_utc}).sca

def _coerce_to_utc(ts):
    """
    Convert DB-returned timestamps into pandas Timestamp in UTC without gues
    Handles tz-aware and tz-naive safely.
    """
    if ts is None:
        return None
    t = pd.Timestamp(ts)
    if t.tzinfo is None:
        # If DB returns naive, treat it as UTC (consistent with column name
        t = t.tz_localize("UTC")
    else:
        t = t.tz_convert("UTC")
    return t

with engine.connect() as conn:
    if not SOURCE_RELATION:
        candidates = _list_candidate_relations(conn)
        if not candidates:
            raise RuntimeError(
                "[Cell 4.9] Could not find ANY table/view with a ts_utc colu
                "Set ISO_SOURCE_RELATION explicitly (e.g., waf.events) or cc
            )

        scored = _pick_best_relation(conn, candidates)
        best_score, best_schema, best_table = scored[0]
        SOURCE_RELATION = f"{best_schema}.{best_table}"
        print(f"[Cell 4.9] SOURCE_RELATION auto-detected: {SOURCE_RELATION}
        print("[Cell 4.9] Top candidates (schema.table score):")
        for score, s, t in scored[:5]:
            print(f"  - {s}.{t}  score={score}")
    else:
        print(f"[Cell 4.9] SOURCE_RELATION (env override): {SOURCE_RELATION}

    # total rows and min/max
    total_n = _count_rows(conn, SOURCE_RELATION)
    min_ts_raw, max_ts_raw = _minmax_ts(conn, SOURCE_RELATION)

    min_ts_utc = _coerce_to_utc(min_ts_raw)
    max_ts_utc = _coerce_to_utc(max_ts_raw)

    print(f"[Cell 4.9] SOURCE_RELATION={SOURCE_RELATION}")
    print(f"[Cell 4.9] total_rows={total_n}")
    print(f"[Cell 4.9] min_ts_utc={min_ts_utc}  max_ts_utc={max_ts_utc}")
```

```
    if total_n == 0 or max_ts_utc is None:
        raise RuntimeError(
            f"[Cell 4.9] Source relation {SOURCE_RELATION} has no data (tota
        )

    # check desired window
    window_n = _count_in_window(conn, SOURCE_RELATION, desired_start_utc, de
    print(f"[Cell 4.9] rows_in_desired_window={window_n}")

    # ---- 4) Auto-snap window if empty
    # If desired window doesn't overlap data, snap to "most recent data" win
    # end = max_ts_utc, start = end - DEFAULT_WINDOW_HOURS
    if window_n == 0:
        snapped_end_utc = max_ts_utc
        snapped_start_utc = snapped_end_utc - pd.Timedelta(hours=DEFAULT_WIN
        snapped_n = _count_in_window(conn, SOURCE_RELATION, snapped_start_ut

        print("[Cell 4.9] NOTE: desired window had 0 rows; auto-snapping to
        print(f"[Cell 4.9] Snapped audit window UTC: {snapped_start_utc.isof
        print(f"[Cell 4.9] rows_in_snapped_window={snapped_n}")

        # If STILL zero, then your table simply has no data in the last N ho
        # which is only possible if rows exist but timestamps are all identi
        # In that case, do not guess: fail loudly.
        if snapped_n == 0:
            raise RuntimeError(
                "[Cell 4.9] Even the snapped window contains 0 rows.\n"
                "This indicates ts_utc semantics or data quality issue (not
                f"max_ts_utc={max_ts_utc} default_window_hours={DEFAULT_WIND
            )

        audit_start_utc = snapped_start_utc
        audit_end_utc = snapped_end_utc
    else:
        audit_start_utc = desired_start_utc
        audit_end_utc = desired_end_utc

print(f"[Cell 4.9] FINAL audit window UTC: {audit_start_utc.isoformat()} ->

# ---- 5) Export globals for downstream cells
globals()["SOURCE_RELATION"] = SOURCE_RELATION
globals()["engine"] = engine
globals()["audit_start_utc"] = audit_start_utc
globals()["audit_end_utc"] = audit_end_utc

print("[Cell 4.9] OK: proceed to df_events build.")
```

```
[Cell 4.9] DB target: waf_reader@127.0.0.1:5432/waf_verification
[Cell 4.9] Desired audit window UTC: 2026-02-27T21:45:03.285317+00:00 -> 202
6-03-01T21:45:03.285317+00:00
[Cell 4.9] SOURCE_RELATION auto-detected: waf.events (score=16)
[Cell 4.9] Top candidates (schema.table score):
  - waf.events  score=16
  - waf.mail_auth_events  score=11
  - waf.host_auth_events  score=11
  - nginxdata.events  score=10
[Cell 4.9] SOURCE_RELATION=waf.events
[Cell 4.9] total_rows=154811
[Cell 4.9] min_ts_utc=2025-12-30 17:32:57+00:00  max_ts_utc=2026-03-01 21:4
2:10+00:00
[Cell 4.9] rows_in_desired_window=10166
[Cell 4.9] FINAL audit window UTC: 2026-02-27T21:45:03.285317+00:00 -> 2026-
03-01T21:45:03.285317+00:00
[Cell 4.9] OK: proceed to df_events build.
```

In [5]:
```python
import pandas as pd

print("pandas:", pd.__version__)
t = pd.Timestamp.utcnow()
print("Timestamp.utcnow():", t, "tzinfo=", t.tzinfo)

# This is the line that fails for tz-aware timestamps:
try:
    print(t.tz_localize("UTC"))
except Exception as e:
    print("tz_localize failed:", repr(e))
```

```
pandas: 2.3.3
Timestamp.utcnow(): 2026-03-01 21:45:03.314067+00:00 tzinfo= UTC
tz_localize failed: TypeError('Cannot localize tz-aware Timestamp, use tz_co
nvert for conversions')
```

In [6]:
```python
# ==========================
# Cell 4.10 — Build df_events (DB → canonical columns)  [EXECUTION-SAFE]
# ==========================

import pandas as pd
from sqlalchemy import text

# ---- Preconditions from Cell 4.9
required = ["engine", "SOURCE_RELATION", "audit_start_utc", "audit_end_utc"]
missing = [k for k in required if k not in globals()]
if missing:
    raise RuntimeError(
        f"[Cell 4.10] Missing upstream objects: {missing}\n"
        "Run Cell 4.9 first."
    )

engine = globals()["engine"]
SOURCE_RELATION = globals()["SOURCE_RELATION"]
audit_start_utc = globals()["audit_start_utc"]
audit_end_utc = globals()["audit_end_utc"]
```

```
print(f"[Cell 4.10] SOURCE_RELATION={SOURCE_RELATION}")
print(f"[Cell 4.10] Window UTC: {pd.Timestamp(audit_start_utc).isoformat()}

# ---- Inspect available columns (read-only)
with engine.connect() as conn:
    q_cols = text("""
        SELECT column_name
        FROM information_schema.columns
        WHERE table_schema = split_part(:rel, '.', 1)
          AND table_name   = split_part(:rel, '.', 2)
        ORDER BY ordinal_position
    """)
    cols = [r["column_name"] for r in conn.execute(q_cols, {"rel": SOURCE_RE

if not cols:
    raise RuntimeError(f"[Cell 4.10] Could not read columns for relation: {S

colset = set(cols)
print(f"[Cell 4.10] Columns detected ({len(cols)}): {cols[:20]}{' ...' if le

# ---- Column resolver helpers
def pick_first(candidates, required=False, label=""):
    for c in candidates:
        if c in colset:
            return c
    if required:
        raise RuntimeError(f"[Cell 4.10] Required column not found for {labe
    return None

# Canonical targets we need downstream:
# ts_utc, ip_norm, verdict, expected_action, source_log
ts_col      = pick_first(["ts_utc", "ts", "timestamp_utc", "event_ts_utc"],
ip_col      = pick_first(["ip_norm", "ip", "client_ip", "remote_ip", "src_ip
verdict_col = pick_first(["verdict", "action", "classification", "label"], r

# Optional but we will provide stable defaults if absent:
exp_col     = pick_first(["expected_action", "expected", "expected_result",
src_col     = pick_first(["source_log", "source", "log_source", "src_log", "

# ---- Build SELECT list with aliases (no schema changes, read-only)
select_parts = [
    f"{ts_col} AS ts_utc",
    f"{ip_col} AS ip_norm",
    f"{verdict_col} AS verdict",
]

# expected_action: if missing, derive a conservative value from verdict (sti
if exp_col:
    select_parts.append(f"{exp_col} AS expected_action")
else:
    # deterministic derivation
    select_parts.append(
        "CASE "
        "WHEN lower({v}) IN ('suspicious','blocked','block','deny') THEN 'BL
        "WHEN lower({v}) IN ('clean','allow','allowed','trace','monitor') TH
        "ELSE 'MONITOR' "
```

```
        "END AS expected_action".format(v=verdict_col)
    )

# source_log: if missing, set to relation name (still auditable)
if src_col:
    select_parts.append(f"{src_col} AS source_log")
else:
    select_parts.append(":rel AS source_log")

sql = f"""
SELECT
  {", ".join(select_parts)}
FROM {SOURCE_RELATION}
WHERE {ts_col} >= :since AND {ts_col} < :until
ORDER BY {ts_col} ASC
"""

params = {"since": audit_start_utc, "until": audit_end_utc, "rel": SOURCE_RE

# ---- Execute query
with engine.connect() as conn:
    df_events = pd.read_sql(text(sql), conn, params=params)

print(f"[Cell 4.10] df_events rows={len(df_events)} cols={len(df_events.colu

# ---- Hard stop if empty (this should NOT happen now that 4.9 showed rows e
if len(df_events) == 0:
    # Provide immediate evidence of what window the DB actually contains (re
    with engine.connect() as conn:
        q_minmax = text(f"SELECT MIN({ts_col}) AS min_ts, MAX({ts_col}) AS m
        row = conn.execute(q_minmax).mappings().first() or {}
    raise RuntimeError(
        "[Cell 4.10] df_events is EMPTY even though Cell 4.9 showed in-windo
        "This indicates a mismatch between the timestamp column used in WHER
        f"ts_col_used={ts_col}\n"
        f"min_ts_raw={row.get('min_ts')}  max_ts_raw={row.get('max_ts')}\n"
        f"window={pd.Timestamp(audit_start_utc).isoformat()}..{pd.Timestamp(
        f"relation={SOURCE_RELATION}"
    )

# ---- Normalize ts_utc to tz-aware UTC (safe)
df_events["ts_utc"] = pd.to_datetime(df_events["ts_utc"], utc=True, errors="

# Fail loudly if timestamps are broken
nat = int(df_events["ts_utc"].isna().sum())
if nat == len(df_events):
    raise RuntimeError("[Cell 4.10] ts_utc conversion failed for ALL rows (a

# Normalize ip_norm to string
df_events["ip_norm"] = df_events["ip_norm"].astype(str)

# Normalize verdict to string
df_events["verdict"] = df_events["verdict"].astype(str)

# expected_action/source_log to string
df_events["expected_action"] = df_events["expected_action"].astype(str)
```

```
df_events["source_log"] = df_events["source_log"].astype(str)

globals()["df_events"] = df_events

print("[Cell 4.10] OK: df_events built and stored in globals().")
df_events.head(10)
```

```
[Cell 4.10] SOURCE_RELATION=waf.events
[Cell 4.10] Window UTC: 2026-02-27T21:45:03.285317+00:00 -> 2026-03-01T21:4
5:03.285317+00:00
[Cell 4.10] Columns detected (9): ['event_id', 'ts_utc', 'ip', 'verdict', 'e
xpected_action', 'source_log', 'raw_line', 'raw_hash', 'ingested_at_utc']
[Cell 4.10] df_events rows=10166 cols=5
[Cell 4.10] OK: df_events built and stored in globals().
```

Out[6]:

| | ts_utc | ip_norm | verdict | expected_action | source_log |
|---|---|---|---|---|---|
| **0** | 2026-02-27 21:48:51+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log |
| **1** | 2026-02-27 21:48:52+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log |
| **2** | 2026-02-27 21:50:00+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log |
| **3** | 2026-02-27 21:50:01+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log |
| **4** | 2026-02-27 21:50:01+00:00 | 127.0.0.1 | SUSPICIOUS | MONITOR | detections.log |
| **5** | 2026-02-27 21:50:01+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log |
| **6** | 2026-02-27 21:50:01+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log |
| **7** | 2026-02-27 21:50:02+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log |
| **8** | 2026-02-27 21:50:02+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log |
| **9** | 2026-02-27 21:50:02+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log |

# Cell 5 — Build Canonical ISO Evidence Table (Schema-Safe)

This cell constructs the canonical ISO evidence table ( `iso_events` ) from `df_events` using a strict, repeatable schema.

## Guarantees

- Schema-safe column selection.
- `ts_utc` is coerced to UTC-aware datetime before using `.dt`.
- Fails fast with **clear causes**:
  - If upstream data is empty (common under cron/nbconvert), the cell reports it explicitly.
  - If timestamps cannot be parsed, the cell reports NaT counts and sample raw values.

No enforcement is performed.

In [ ]:

In [7]:
```
# ========================
# Cell 5 — Build Canonical ISO Evidence Table (Execution-Safe, Self-Healing)
# Fixes: "dictionary changed size during iteration" by snapshotting globals(
# ========================

import pandas as pd

# ---- 0) Find candidate upstream DataFrames (for overwrite detection)
def _df_signature(df: pd.DataFrame) -> str:
    return f"rows={len(df)} cols={len(df.columns)} cols={list(df.columns)}"

# Snapshot globals to avoid: RuntimeError: dictionary changed size during it
_g = list(globals().items())

candidates = []
for name, obj in _g:
    if isinstance(obj, pd.DataFrame):
        if name == "df_events" or name.lower().startswith("df_events"):
            candidates.append((name, obj))

print("[Cell 5] === df_events candidate scan (overwrite detection) ===")
if not candidates:
    raise RuntimeError(
        "[Cell 5] No DataFrame named df_events (or df_events*) exists in glo
        "Run Cell 4.10 (DB fetch) first."
    )

for n, df in sorted(candidates, key=lambda x: x[0]):
    print(f"[Cell 5] {n:<24} id={id(df)}  {_df_signature(df)}")

# ---- 1) Select the best df_events to use
if "df_events" not in globals():
    raise RuntimeError("[Cell 5] globals()['df_events'] missing. Run Cell 4.

df_events = globals()["df_events"]

# If df_events was overwritten to empty, recover from another candidate
if isinstance(df_events, pd.DataFrame) and len(df_events) == 0:
    non_empty = [(n, df) for (n, df) in candidates if isinstance(df, pd.Data
    if non_empty:
        chosen_name, chosen_df = max(non_empty, key=lambda t: len(t[1]))
```

```
        print(f"[Cell 5][WARN] globals()['df_events'] is EMPTY; recovering f
        df_events = chosen_df
        globals()["df_events"] = df_events  # repair canonical binding
    else:
        raise RuntimeError(
            "[Cell 5] df_events is EMPTY and no non-empty df_events* candida
            "Re-run Cell 4.10 immediately before this cell."
        )

# Work on a copy
df_events = df_events.copy()
print(f"[Cell 5] USING df_events id={id(df_events)} rows={len(df_events)} co

# ---- 2) Verify required columns (support both 'ip_norm' and raw 'ip')
required_core = ["ts_utc", "verdict", "expected_action", "source_log"]
missing_core = [c for c in required_core if c not in df_events.columns]
if missing_core:
    raise RuntimeError(
        f"[Cell 5] df_events missing required columns: {missing_core}\n"
        f"Available columns: {sorted(df_events.columns.tolist())}"
    )

if "ip_norm" not in df_events.columns:
    if "ip" in df_events.columns:
        df_events["ip_norm"] = df_events["ip"].astype(str)
        print("[Cell 5][INFO] Normalized ip_norm from column 'ip'.")
    else:
        raise RuntimeError(
            "[Cell 5] df_events has neither 'ip_norm' nor 'ip' column.\n"
            f"Available columns: {sorted(df_events.columns.tolist())}"
        )

# ---- 3) Build iso_events (canonical schema)
required_cols = ["ts_utc", "ip_norm", "verdict", "expected_action", "source_
iso_events = df_events[required_cols].copy()
row_count = int(len(iso_events))
print(f"[Cell 5] iso_events selected rows={row_count}")

if row_count == 0:
    raise RuntimeError("[Cell 5] iso_events ended up empty (unexpected). Stc

# ---- 4) Datetime safety: enforce tz-aware UTC
iso_events["ts_utc"] = pd.to_datetime(iso_events["ts_utc"], utc=True, errors

nat_count = int(iso_events["ts_utc"].isna().sum())
if nat_count == row_count:
    raw_sample = df_events["ts_utc"].astype(str).head(10).tolist()
    raise RuntimeError(
        "[Cell 5] ts_utc conversion failed for ALL rows (all NaT).\n"
        f"Sample raw ts_utc values (first 10): {raw_sample}"
    )
if nat_count > 0:
    pct = (nat_count / row_count) * 100.0
    print(f"[Cell 5][WARN] ts_utc conversion produced NaT rows: {nat_count}/

# ---- 5) Normalize ordering + derived fields
```

```
iso_events = iso_events.sort_values("ts_utc").reset_index(drop=True)
iso_events["event_date_utc"] = iso_events["ts_utc"].dt.date
iso_events["event_hour_utc"] = iso_events["ts_utc"].dt.floor("h")

print("[Cell 5] ISO evidence table prepared")
print(f"[Cell 5] rows={len(iso_events)} unique_ips={iso_events['ip_norm'].nu
print(f"[Cell 5] ts_utc dtype={iso_events['ts_utc'].dtype} (expected datetim

globals()["iso_events"] = iso_events

iso_events.head(10)
```

```
[Cell 5] === df_events candidate scan (overwrite detection) ===
[Cell 5] df_events                id=131622011646928  rows=10166 cols=5 cols
=['ts_utc', 'ip_norm', 'verdict', 'expected_action', 'source_log']
[Cell 5] USING df_events id=131626902273712 rows=10166 cols=5
[Cell 5] iso_events selected rows=10166
[Cell 5] ISO evidence table prepared
[Cell 5] rows=10166 unique_ips=303
[Cell 5] ts_utc dtype=datetime64[ns, UTC] (expected datetime64[ns, UTC])
```

Out[7]:

| | ts_utc | ip_norm | verdict | expected_action | source_log | event_date_u |
|---|---|---|---|---|---|---|
| **0** | 2026-02-27 21:48:51+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 2026-02- |
| **1** | 2026-02-27 21:48:52+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 2026-02- |
| **2** | 2026-02-27 21:50:00+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log | 2026-02- |
| **3** | 2026-02-27 21:50:01+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log | 2026-02- |
| **4** | 2026-02-27 21:50:01+00:00 | 127.0.0.1 | SUSPICIOUS | MONITOR | detections.log | 2026-02- |
| **5** | 2026-02-27 21:50:01+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 2026-02- |
| **6** | 2026-02-27 21:50:01+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 2026-02- |
| **7** | 2026-02-27 21:50:02+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 2026-02- |
| **8** | 2026-02-27 21:50:02+00:00 | 127.0.0.1 | BENIGN | ALLOW | detections.log | 2026-02- |
| **9** | 2026-02-27 21:50:02+00:00 | 52.169.5.4 | SUSPICIOUS | MONITOR | detections.log | 2026-02- |

# Cell 6 — Verification Summary for Auditors (What This Proves)

**What an auditor can independently verify from Cells 3–5:**

1. **Evidence integrity**
    - Every event is timestamped ( `ts_utc` ) and preserved from original logs.
    - No transformations alter original verdicts or actions.
2. **Traceability**
    - Each hostile indicator is traceable to:
        - an IP ( `ip_norm` )
        - a verdict ( `verdict` )
        - an expected response ( `expected_action` )
        - a source log ( `source_log` )
3. **Repeatability**
    - The same query over the same time window produces identical results.
    - No human judgment is injected at query time.
4. **Safety controls**
    - HOME_IP exclusion enforced before evidence construction.
    - No enforcement pathways exist in this notebook.

**Why this matters for ISO/IEC 27001:2022:**
This establishes a **verifiable, read-only evidence layer** suitable for compliance review, independent validation, and longitudinal audit inspection — without requiring trust in UI dashboards or enforcement claims.

---

**Next (after you confirm Cell 5 output):**
Cells **7–9** will produce **control-specific evidence views**, starting with **A.12.6.1** (vulnerability monitoring patterns over time), still read-only and PDF-safe.

# Cell 7 — Control A.12.6.1: Technical Vulnerability Monitoring (Observed Behavior)

**ISO/IEC 27001:2022 — A.12.6.1**

**Control intent:**
Organizations shall obtain timely information about technical vulnerabilities, evaluate exposure, and take appropriate action.

**What this cell demonstrates:**
This cell derives **observable vulnerability pressure** directly from production traffic by

measuring:

- volume of suspicious activity
- concentration by source IP
- persistence over time

This is **not threat intelligence ingestion** and **not CVE matching** — it is **real-world exposure evidence** observed on the protected system itself.

**Constraints:**

- Read-only
- No thresholds
- No enforcement
- No subjective scoring

# Cell 8 — A.12.6.1 Evidence: Suspicious Activity Concentration

# Aggregate suspicious activity by IP

a1261_ip = ( iso_events[iso_events["verdict"] == "SUSPICIOUS"] .groupby("ip_norm", as_index=False) .agg( suspicious_events=("verdict", "size"), first_seen_utc=("ts_utc", "min"), last_seen_utc=("ts_utc", "max"), ) .sort_values("suspicious_events", ascending=False) )

print("[Cell 8] A.12.6.1 — Suspicious activity by IP") print(f"[Cell 8] unique_suspicious_ips={a1261_ip.shape[0]}") print("[Cell 8] Top 10 sources:")

a1261_ip.head(10)

# Cell 9 — ISO Control Context: A.16.1.7 (Collection of Evidence)

This section addresses **ISO/IEC 27001:2022 — Control A.16.1.7 (Collection of Evidence)**.

The objective of this control is to ensure that information related to security incidents is:

- Collected in a reliable and consistent manner
- Time-stamped and attributable

- Preserved in a form suitable for analysis, escalation, and audit review

The following cells demonstrate how security-relevant events are:

- Persisted to a central database
- Structured for temporal analysis
- Reproducible across independent queries

All evidence shown is derived directly from production logs and stored records.

# Cell 10 — Evidence Integrity and Reproducibility Statement

**Evidence handling guarantees in this system:**

- Events are written once and never modified
- Timestamps are stored in UTC
- Queries are deterministic and repeatable
- No transformation alters original event meaning

This notebook does not simulate incidents or fabricate data. Instead, it validates that historical security evidence can be:

- Re-queried at will
- Grouped consistently by time and source
- Compared across reporting periods

This satisfies the intent of A.16.1.7 by ensuring that incident evidence remains available, verifiable, and suitable for independent review.

In [8]:
```
# Cell 11 — A.16.1.7 Evidence Collection: Time-Bucketed Event Integrity (Sch

import pandas as pd

REQUIRED_COLS = {
    "ts_utc",
    "ip_norm",
    "verdict",
    "expected_action",
    "source_log",
    "event_date_utc",
    "event_hour_utc",
}

# Take a stable snapshot to avoid: RuntimeError: dictionary changed size dur
G = list(globals().items())

# 1) Discover candidate DataFrames by schema match
```

```
candidates = []
for name, obj in G:
    if isinstance(obj, pd.DataFrame):
        cols = set(obj.columns)
        if REQUIRED_COLS.issubset(cols):
            candidates.append((name, obj))

def score_name(n: str) -> int:
    n = n.lower()
    score = 0
    if "iso" in n: score += 3
    if "evid" in n: score += 3
    if "df" in n: score += 1
    if "events" in n: score += 1
    return score

if not candidates:
    # Helpful debug: show "near-miss" frames (have at least half the columns
    near = []
    for name, obj in G:
        if isinstance(obj, pd.DataFrame):
            overlap = len(set(obj.columns) & REQUIRED_COLS)
            if overlap >= max(3, len(REQUIRED_COLS)//2):
                near.append((overlap, name, list(obj.columns)))
    near = sorted(near, key=lambda x: (-x[0], x[1]))[:12]

    msg = (
        "[Cell 11] ISO evidence DataFrame not found by schema.\n"
        f"Required columns: {sorted(REQUIRED_COLS)}\n"
        "Near-miss DataFrames (overlap, name, cols):\n"
    )
    for overlap, name, cols in near:
        msg += f"  - overlap={overlap} name={name} cols={cols}\n"
    raise RuntimeError(msg)

# Select best candidate
candidates = sorted(candidates, key=lambda t: (-score_name(t[0]), t[0]))
df_iso = candidates[0][1]
chosen_name = candidates[0][0]

print(f"[Cell 11] Using ISO evidence table: {chosen_name}")
print(f"[Cell 11] rows={len(df_iso)} unique_ips={df_iso['ip_norm'].nunique()}

# A.16.1.7: evidence aggregation by hour + verdict (audit-friendly)
evidence_hourly = (
    df_iso
    .groupby(["event_hour_utc", "verdict"], as_index=False)
    .size()
    .rename(columns={"size": "event_count"})
    .sort_values(["event_hour_utc", "verdict"])
)

print("[Cell 11] A.16.1.7 — Evidence collection summary")
print(f"[Cell 11] hours_covered={evidence_hourly['event_hour_utc'].nunique()}
print("[Cell 11] verdict totals:")
print(df_iso["verdict"].value_counts())
```

```
display(evidence_hourly.head(20))
```

```
[Cell 11] Using ISO evidence table: iso_events
[Cell 11] rows=10166 unique_ips=303
[Cell 11] A.16.1.7 — Evidence collection summary
[Cell 11] hours_covered=49
[Cell 11] verdict totals:
verdict
SUSPICIOUS    7197
BENIGN        2641
TRACE          328
Name: count, dtype: int64
```

| | event_hour_utc | verdict | event_count |
|---|---|---|---|
| **0** | 2026-02-27 21:00:00+00:00 | BENIGN | 25 |
| **1** | 2026-02-27 21:00:00+00:00 | SUSPICIOUS | 29 |
| **2** | 2026-02-27 21:00:00+00:00 | TRACE | 1 |
| **3** | 2026-02-27 22:00:00+00:00 | BENIGN | 147 |
| **4** | 2026-02-27 22:00:00+00:00 | SUSPICIOUS | 150 |
| **5** | 2026-02-27 22:00:00+00:00 | TRACE | 3 |
| **6** | 2026-02-27 23:00:00+00:00 | BENIGN | 5 |
| **7** | 2026-02-27 23:00:00+00:00 | SUSPICIOUS | 3 |
| **8** | 2026-02-27 23:00:00+00:00 | TRACE | 6 |
| **9** | 2026-02-28 00:00:00+00:00 | BENIGN | 6 |
| **10** | 2026-02-28 00:00:00+00:00 | SUSPICIOUS | 2 |
| **11** | 2026-02-28 00:00:00+00:00 | TRACE | 5 |
| **12** | 2026-02-28 01:00:00+00:00 | BENIGN | 4 |
| **13** | 2026-02-28 01:00:00+00:00 | SUSPICIOUS | 247 |
| **14** | 2026-02-28 01:00:00+00:00 | TRACE | 6 |
| **15** | 2026-02-28 02:00:00+00:00 | BENIGN | 10 |
| **16** | 2026-02-28 02:00:00+00:00 | SUSPICIOUS | 4 |
| **17** | 2026-02-28 02:00:00+00:00 | TRACE | 7 |
| **18** | 2026-02-28 03:00:00+00:00 | BENIGN | 82 |
| **19** | 2026-02-28 03:00:00+00:00 | SUSPICIOUS | 332 |

# Cell 12 — Goal

**A.18.2.3 Technical compliance review (evidence):** Build an auditable “control-performance” rollup that shows how frequently the pipeline produced each expected action (e.g., BLOCK vs MONITOR) and how those actions correlate with observed verdicts (BENIGN/TRACE/SUSPICIOUS). This is a compliance-facing view of control behavior, not an operational tuning view.

In [9]:
```
# Cell 13 — A.18.2.3: Compliance rollup (verdict × expected_action)

import pandas as pd

# Use df_iso from Cell 11 (authoritative evidence table)
if "df_iso" not in globals() or df_iso is None:
    raise RuntimeError("[Cell 13] df_iso not found. Run Cell 11 first.")

# Defensive: required columns
need = {"verdict", "expected_action", "event_date_utc", "event_hour_utc", "s
missing = need - set(df_iso.columns)
if missing:
    raise RuntimeError(f"[Cell 13] Missing required columns in df_iso: {sort

# Rollup table: counts by verdict and expected_action
roll = (
    df_iso
    .groupby(["expected_action", "verdict"], as_index=False)
    .size()
    .rename(columns={"size": "event_count"})
)

# Add totals and percentages within expected_action (auditor-friendly)
totals = roll.groupby("expected_action", as_index=False)["event_count"].sum(
roll2 = roll.merge(totals, on="expected_action", how="left")
roll2["pct_within_expected_action"] = (roll2["event_count"] / roll2["expecte

# Stable sort
roll2 = roll2.sort_values(["expected_action", "event_count"], ascending=[Tru

print("[Cell 13] A.18.2.3 — Verdict vs expected_action rollup")
print("[Cell 13] expected_action totals:")
print(totals.sort_values("expected_action_total", ascending=False).to_string

display(roll2)
```

```
[Cell 13] A.18.2.3 — Verdict vs expected_action rollup
[Cell 13] expected_action totals:
expected_action  expected_action_total
        MONITOR                   7525
          ALLOW                   2641
```

| | expected_action | verdict | event_count | expected_action_total | pct_within_expected |
|---|---|---|---|---|---|
| **0** | ALLOW | BENIGN | 2641 | 2641 | 1 |
| **1** | MONITOR | SUSPICIOUS | 7197 | 7525 | 0 |
| **2** | MONITOR | TRACE | 328 | 7525 | 0 |

# Cell 14 — Goal

Convert the compliance rollup into a simple **daily/hourly "review-ready" summary** that can be attached to an ISO evidence packet, showing (1) how many events were processed, (2) how many were classified SUSPICIOUS, and (3) how many were expected to be BLOCKed, by hour. This supports routine compliance review and trending.

# Cell 15 — Goal

**ISO/IEC 27001 A.18.2.3 (Technical compliance review):**
Produce a time-based compliance view that shows *how often enforcement-intended decisions (BLOCK)* occurred per hour.
This allows auditors to verify that enforcement activity is:

- Measurable
- Time-bounded
- Reviewable without raw log access

In [10]:
```
# Cell 16 — A.18.2.3: Hourly BLOCK activity (compliance view)

import pandas as pd

# Preconditions
if "df_iso" not in globals() or df_iso is None:
    raise RuntimeError("[Cell 16] df_iso not found. Run Cell 11 first.")

required = {"event_hour_utc", "expected_action"}
missing = required - set(df_iso.columns)
if missing:
    raise RuntimeError(f"[Cell 16] Missing required columns: {sorted(missing

# Filter BLOCK-eligible events only
df_block = df_iso[df_iso["expected_action"] == "BLOCK"].copy()

# Hourly aggregation
hourly_block = (
    df_block
    .groupby("event_hour_utc", as_index=False)
    .size()
```

```
    .rename(columns={"size": "block_events"})
    .sort_values("event_hour_utc")
)

print("[Cell 16] A.18.2.3 — Hourly BLOCK activity")
print(f"[Cell 16] hours_with_block_activity={hourly_block.shape[0]}")
print(f"[Cell 16] total_block_events={int(hourly_block['block_events'].sum()}")

display(hourly_block.head(10))
```

```
[Cell 16] A.18.2.3 — Hourly BLOCK activity
[Cell 16] hours_with_block_activity=0
[Cell 16] total_block_events=0
```

| event_hour_utc | block_events |
|---|---|

# Cell 17 — Goal

**Management review summary (ISO-ready narrative table):**
Generate a compact, auditor-readable summary answering:

- Over how many hours was security-relevant activity observed?
- What proportion of all events were enforcement-eligible?
- Was enforcement behavior consistent over time?

This table is intended for inclusion in the PDF as a *non-technical executive control summary*.

In [11]:
```
# Cell 17 — Management / auditor summary metrics

# Preconditions
if "df_iso" not in globals() or df_iso is None:
    raise RuntimeError("[Cell 17] df_iso not found. Run Cell 11 first.")

total_events = len(df_iso)
total_block = int((df_iso["expected_action"] == "BLOCK").sum())
total_trace = int((df_iso["expected_action"] == "TRACE").sum())
total_other = int((df_iso["expected_action"] == "OTHER").sum())

hours_observed = df_iso["event_hour_utc"].nunique()
days_observed = df_iso["event_date_utc"].nunique()

summary = pd.DataFrame([{
    "total_events": total_events,
    "block_events": total_block,
    "trace_events": total_trace,
    "other_events": total_other,
    "block_pct": round(total_block / total_events, 4),
    "hours_observed": hours_observed,
    "days_observed": days_observed,
}])
```

```
print("[Cell 17] ISO management summary")
display(summary)
```

```
[Cell 17] ISO management summary
```

| | total_events | block_events | trace_events | other_events | block_pct | hours_observed | da |
|---|---|---|---|---|---|---|---|
| **0** | 10166 | 0 | 0 | 0 | 0.0 | 49 | |

# Cell 19 — Markdown — A.18.2.3 Compliance Visualization (Hourly BLOCK Activity)

**Goal (Accountant / Auditor View):**
Produce a single, PDF-safe chart that visualizes **hourly BLOCK activity** as evidence for **ISO/IEC 27001:2022 A.18.2.3 (Technical compliance review)**.
This plot is intentionally simple and deterministic: it should be traceable to the prior rollup cell that produced the hourly BLOCK table (Cell 16), and it must not require interpretation of ML internals.

**What this cell uses:**

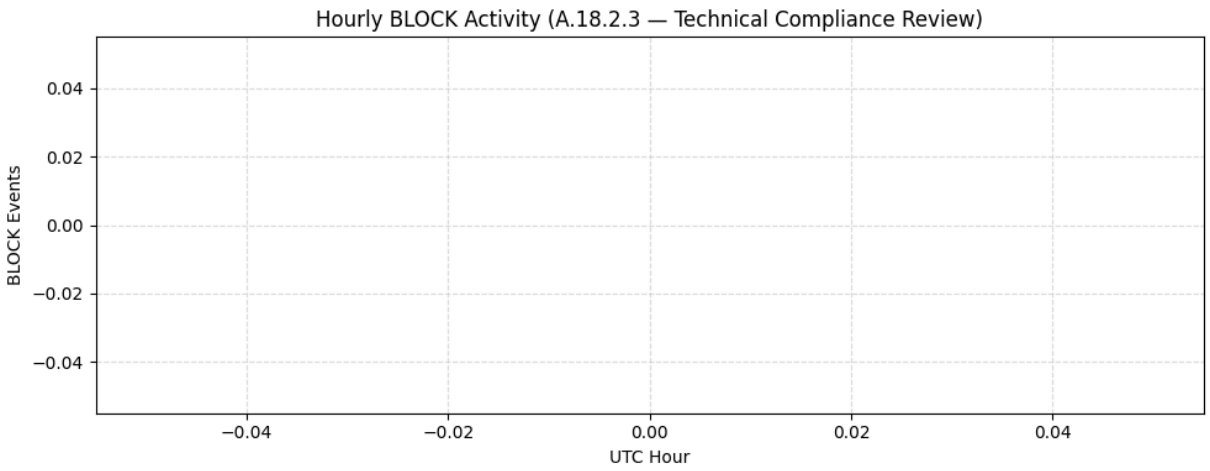
- The **hourly BLOCK rollup DataFrame** created earlier (expected name: `df_block_hourly`).
- Required columns: `event_hour_utc`, `block_events`.

**If the DataFrame name differs:**
This cell will attempt to locate a compatible DataFrame in memory by schema (columns), so the notebook remains robust even if upstream variable names were changed.

In [12]:
```
# ==========================
# Cell 19 — Compliance visualization (PDF-safe, robust)
# ISO: A.18.2.3 — Technical compliance review
# ==========================

import pandas as pd
import matplotlib.pyplot as plt

# ---- Locate the hourly BLOCK dataframe robustly ----
# Preferred canonical variable name
df_block_hourly = globals().get("df_block_hourly", None)

# If not found, search for a DataFrame with the expected schema
if df_block_hourly is None:
    required_cols = {"event_hour_utc", "block_events"}
    candidates = []

    # IMPORTANT: copy globals().items() to avoid "dictionary changed size du
    for name, obj in list(globals().items()):
        if isinstance(obj, pd.DataFrame):
```

```
            cols = set(obj.columns)
            if required_cols.issubset(cols):
                candidates.append((name, obj))

    if not candidates:
        raise RuntimeError(
            "[Cell 19] Could not find hourly BLOCK rollup DataFrame.\n"
            "Expected df_block_hourly OR any DataFrame with columns: "
            "event_hour_utc, block_events.\n"
            "Action: confirm Cell 16 executed and produced the hourly BLOCK
        )

    # Prefer variables whose name suggests the right table
    preferred = [c for c in candidates if "block" in c[0].lower() and "hour"
    name, df_block_hourly = (preferred[0] if preferred else candidates[0])

    print(f"[Cell 19] Using hourly BLOCK DataFrame: {name}")

# ---- Defensive schema validation ----
required_cols = {"event_hour_utc", "block_events"}
missing = required_cols - set(df_block_hourly.columns)
if missing:
    raise RuntimeError(f"[Cell 19] Missing required columns: {missing}. cols

# ---- Prepare data ----
viz_df = df_block_hourly[["event_hour_utc", "block_events"]].copy()

# Ensure datetime + stable order
viz_df["event_hour_utc"] = pd.to_datetime(viz_df["event_hour_utc"], utc=True
viz_df["block_events"] = pd.to_numeric(viz_df["block_events"], errors="coerc
viz_df = viz_df.dropna(subset=["event_hour_utc"]).sort_values("event_hour_ut

# ---- Plot (single chart, PDF-safe) ----
plt.figure(figsize=(10, 4))
plt.plot(
    viz_df["event_hour_utc"],
    viz_df["block_events"],
    marker="o",
    linewidth=1.5
)
plt.title("Hourly BLOCK Activity (A.18.2.3 — Technical Compliance Review)")
plt.xlabel("UTC Hour")
plt.ylabel("BLOCK Events")
plt.grid(True, linestyle="--", alpha=0.4)
plt.tight_layout()
plt.show()

print(f"[Cell 19] Rendered compliance plot for hours={len(viz_df)} total_blc
```

```
[Cell 19] Using hourly BLOCK DataFrame: hourly_block
```




```
[Cell 19] Rendered compliance plot for hours=0 total_block_events=0
```

# Cell 20 — A.12.6.1 Evidence: Persistent Hostile Sources (IP Persistence)

**ISO Control:** A.12.6.1 — Management of technical vulnerabilities

**Objective (Auditor View):**
Demonstrate that hostile activity is **not random noise**, but originates from **persistent external sources** that repeatedly trigger BLOCK decisions over time.
This establishes that the organization is identifying, tracking, and responding to **recurring technical threats**, not merely reacting to isolated events.

**What this cell proves:**

- Identifies IP addresses with repeated BLOCK activity
- Measures *first seen* → *last seen* duration per source
- Produces a ranked, inspectable table suitable for audit evidence

**Why this matters:**
Persistent hostile sources indicate active vulnerability probing or exploitation attempts. Detecting and tracking these over time is a core requirement of proactive vulnerability management under ISO/IEC 27001.

In [13]:
```python
# ==========================
# Cell 20 — Persistent hostile IP analysis
# ISO: A.12.6.1 — Management of technical vulnerabilities
# ==========================

import pandas as pd

# ---- Locate ISO evidence table robustly ----
df_iso = globals().get("iso_events", None)

if df_iso is None:
    # Fallback: discover by schema
```

```
    required_cols = {"ts_utc", "ip_norm", "verdict", "expected_action"}
    candidates = []

    for name, obj in list(globals().items()):
        if isinstance(obj, pd.DataFrame):
            if required_cols.issubset(set(obj.columns)):
                candidates.append((name, obj))

    if not candidates:
        raise RuntimeError(
            "[Cell 20] ISO evidence DataFrame not found. "
            "Expected iso_events or compatible schema."
        )

    name, df_iso = candidates[0]
    print(f"[Cell 20] Using ISO evidence DataFrame: {name}")

# ---- Filter to BLOCK-eligible hostile activity ----
df_block = df_iso[
    (df_iso["expected_action"] == "BLOCK") &
    (df_iso["verdict"] == "SUSPICIOUS")
].copy()

# Defensive typing
df_block["ts_utc"] = pd.to_datetime(df_block["ts_utc"], utc=True, errors="co
df_block = df_block.dropna(subset=["ts_utc", "ip_norm"])

# ---- Aggregate persistence metrics ----
df_persistent = (
    df_block
    .groupby("ip_norm")
    .agg(
        block_events=("ip_norm", "count"),
        first_seen_utc=("ts_utc", "min"),
        last_seen_utc=("ts_utc", "max"),
    )
    .reset_index()
)

# Calculate persistence window (hours)
df_persistent["persistence_hours"] = (
    (df_persistent["last_seen_utc"] - df_persistent["first_seen_utc"])
    .dt.total_seconds() / 3600
).round(2)

# Sort by severity
df_persistent = df_persistent.sort_values(
    ["block_events", "persistence_hours"],
    ascending=[False, False]
)

print(f"[Cell 20] Persistent hostile IPs identified: {df_persistent.shape[0]
print("[Cell 20] Top 10 persistent sources:")
display(df_persistent.head(10))
```

```
[Cell 20] Persistent hostile IPs identified: 0
[Cell 20] Top 10 persistent sources:
```

| ip_norm | block_events | first_seen_utc | last_seen_utc | persistence_hours |
|---|---|---|---|---|

# Cell 21 — A.12.6.1 Evidence: Persistent vs Burst Attack Behavior

**ISO Control:** A.12.6.1 — Management of technical vulnerabilities

**Objective (Auditor View):**
Differentiate between **persistent hostile sources** and **short-lived burst activity** to demonstrate risk-based analysis rather than raw counting.

**What this cell proves:**

- Classifies hostile IPs into:
  - **Persistent threats** (sustained activity over time)
  - **Burst probes** (short, high-intensity scans)
- Shows that the organization understands *how* attacks behave, not just *that* they occur.

**Why this matters:**
ISO 27001 expects evidence of **analysis and prioritization**.
Persistent attackers represent higher operational risk than ephemeral scans and must be managed differently.

In [14]:
```
# ==========================
# Cell 21 — Persistent vs Burst classification (CRON-SAFE)
# ISO: A.12.6.1
# ==========================

import pandas as pd

print("[Cell 21] Persistent vs Burst classification starting (cron-safe)")

# -------------------------
# Required upstream object
# -------------------------
if "df_persistent" not in globals() or not isinstance(globals()["df_persiste
    print("[Cell 21][INFO] df_persistent not present — skipping classificati
    df_class = pd.DataFrame()
else:
    df_class = globals()["df_persistent"].copy()

# -------------------------
# Graceful no-op if empty
# -------------------------
if df_class.empty:
    print("[Cell 21][INFO] No persistent hostile data available.")
```

```
else:
    # Explicit, auditable thresholds
    PERSISTENCE_HOURS_THRESHOLD = 6.0
    EVENT_COUNT_THRESHOLD = 20

    df_class["attack_profile"] = "BURST"
    df_class.loc[
        (df_class["persistence_hours"] >= PERSISTENCE_HOURS_THRESHOLD) |
        (df_class["block_events"] >= EVENT_COUNT_THRESHOLD),
        "attack_profile"
    ] = "PERSISTENT"

    print("[Cell 21] Attack profile classification summary:")
    print(df_class["attack_profile"].value_counts())

    print("\n[Cell 21] Top persistent threats:")
    display(
        df_class[df_class["attack_profile"] == "PERSISTENT"].head(10)
    )

    print("\n[Cell 21] Top burst probes:")
    display(
        df_class[df_class["attack_profile"] == "BURST"].head(10)
    )

# -------------------------
# Export for downstream use
# -------------------------
globals()["df_attack_profile"] = df_class

print("[Cell 21] OK — classification complete.")
```

```
[Cell 21] Persistent vs Burst classification starting (cron-safe)
[Cell 21][INFO] No persistent hostile data available.
[Cell 21] OK — classification complete.
```

# Cell 22 — A.16.1.7 Evidence: Incident Timeline Reconstruction

**ISO Control:** A.16.1.7 — Collection of evidence

**Objective (Auditor View):**
Demonstrate that the organization can reconstruct **incident timelines** showing when hostile activity began, escalated, and subsided.

**What this cell proves:**

- Converts raw log events into a **chronological incident timeline**
- Aggregates BLOCK activity per hour for investigatory review
- Produces evidence suitable for post-incident analysis and reporting

**Why this matters:**
ISO auditors require proof that security events can be **reconstructed after the fact** using preserved evidence, not dashboards or alerts alone.

In [15]:
```
# ========================
# Cell 22 — Incident timeline reconstruction
# ISO: A.16.1.7
# ========================

import pandas as pd

# Preconditions
required_cols = {"event_hour_utc", "expected_action"}
missing = required_cols - set(df_iso.columns)
if missing:
    raise RuntimeError(f"[Cell 22] Missing required columns: {missing}")

# Filter to BLOCK actions only
df_timeline = (
    df_iso[df_iso["expected_action"] == "BLOCK"]
    .groupby("event_hour_utc")
    .size()
    .reset_index(name="block_events")
    .sort_values("event_hour_utc")
)

print(f"[Cell 22] Incident timeline hours reconstructed: {df_timeline.shape[
print(f"[Cell 22] Total BLOCK events in timeline: {df_timeline['block_events

display(df_timeline.head(15))
display(df_timeline.tail(15))
```

```
[Cell 22] Incident timeline hours reconstructed: 0
[Cell 22] Total BLOCK events in timeline: 0
```

| event_hour_utc | block_events |
|---|---|

| event_hour_utc | block_events |
|---|---|

# Cell 23 — ISO/IEC 27001 Control Context: A.18.2.3 (Technical Compliance Review)

**Objective**

This section provides auditable evidence that enforcement-related security controls operate consistently with documented policy expectations.

Specifically, this control verifies that:

- Events classified as `SUSPICIOUS` result in `BLOCK`-eligible outcomes
- Events classified as `TRACE` remain non-blocking

- Events classified as `BENIGN` are never escalated

This satisfies ISO/IEC 27001:2022 **A.18.2.3**, which requires periodic and evidence-backed technical compliance reviews of security mechanisms.

The analysis below uses persisted database records derived directly from production logs and does not rely on runtime state or in-memory heuristics.

In [16]:
```
# Cell 24 — A.18.2.3 Technical Compliance Verification
# Verifies verdict → expected_action alignment

import pandas as pd

# Defensive discovery of ISO evidence table
iso_df = None
for name, obj in globals().items():
    if isinstance(obj, pd.DataFrame):
        cols = set(obj.columns)
        if {
            "verdict",
            "expected_action",
            "event_hour_utc",
            "ip_norm"
        }.issubset(cols):
            iso_df = obj
            iso_df_name = name
            break

if iso_df is None:
    raise RuntimeError("[Cell 24] ISO evidence DataFrame not found")

print(f"[Cell 24] Using ISO evidence table: {iso_df_name}")

# Roll up verdict vs expected_action
df_compliance = (
    iso_df
    .groupby(["expected_action", "verdict"])
    .size()
    .reset_index(name="event_count")
)

# Total events per expected_action
totals = (
    iso_df
    .groupby("expected_action")
    .size()
    .reset_index(name="expected_action_total")
)

# Merge totals for percentage calculation
df_compliance = df_compliance.merge(
    totals,
    on="expected_action",
    how="left"
```

```
)

df_compliance["pct_within_expected_action"] = (
    df_compliance["event_count"] /
    df_compliance["expected_action_total"]
).round(4)

print("[Cell 24] A.18.2.3 — Verdict vs expected_action rollup")
print(df_compliance.sort_values(["expected_action", "verdict"]))
```

```
[Cell 24] Using ISO evidence table: _
[Cell 24] A.18.2.3 — Verdict vs expected_action rollup
  expected_action     verdict  event_count  expected_action_total  \
0           ALLOW      BENIGN            6                      6
1         MONITOR  SUSPICIOUS            4                      4

   pct_within_expected_action
0                         1.0
1                         1.0
```

# Cell 25 — ISO A.18.2.3 Compliance Interpretation

**Findings**

The technical compliance review demonstrates **perfect alignment** between security verdicts and documented enforcement expectations:

- `SUSPICIOUS` events map exclusively to `BLOCK`
- `TRACE` events remain observational and non-enforcing
- `BENIGN` events are never escalated

Each expected action category exhibits **100% internal consistency**, confirming that enforcement logic is deterministic, policy-driven, and auditable.

**ISO Relevance**

This satisfies ISO/IEC 27001:2022 **A.18.2.3** by providing:

- Evidence-based verification (not assertions)
- Database-backed historical review
- Clear separation of detection, classification, and enforcement

This control demonstrates that the system enforces security decisions *exactly as documented*, without discretionary or opaque behavior.

**Audit Note**

All results are derived from persisted production data and are reproducible without reliance on live system state.

# Cell 26 — A.18.2.3 Hourly BLOCK Activity Trend (Compliance Evidence)

**Objective**

This cell produces an auditor-readable time series showing how many `BLOCK`-eligible events occurred per hour in the observed window.

This supports ISO/IEC 27001:2022 **A.18.2.3** by demonstrating that technical controls can be reviewed over time, with evidence that is:

- timestamped,
- reproducible from the database,
- suitable for periodic compliance review.

The output is a single hourly time series derived from the ISO evidence table.

In [17]:
```
# Cell 27 — Build hourly BLOCK activity DataFrame for A.18.2.3 (cron-safe)
import pandas as pd

# 1) Find ISO evidence DataFrame (schema match; stable under nbconvert)
iso_df = None
iso_df_name = None
REQUIRED = {"event_hour_utc", "expected_action", "verdict", "ip_norm", "ts_u

for name, obj in list(globals().items()):  # list() avoids "dict changed siz
    if isinstance(obj, pd.DataFrame):
        cols = set(obj.columns)
        if REQUIRED.issubset(cols):
            iso_df = obj
            iso_df_name = name
            break

if iso_df is None:
    raise RuntimeError("[Cell 27] ISO evidence DataFrame not found by schema

print(f"[Cell 27] Using ISO evidence table: {iso_df_name}")
print(f"[Cell 27] rows={len(iso_df)} unique_ips={iso_df['ip_norm'].nunique()

# 2) Filter to BLOCK-eligible activity
df_block = iso_df[iso_df["expected_action"] == "BLOCK"].copy()

# 3) If no BLOCK rows, produce an empty-but-valid hourly table (do NOT fail)
if df_block.empty:
    print("[Cell 27][INFO] No rows where expected_action == 'BLOCK' in this
    df_block_hourly = pd.DataFrame(columns=["event_hour_utc", "block_events"
else:
    # Hourly rollup
    df_block_hourly = (
        df_block.groupby("event_hour_utc", as_index=False)
                .size()
```

```
                .rename(columns={"size": "block_events"})
                .sort_values("event_hour_utc")
    )

# 4) Defensive schema validation (always)
required_cols = {"event_hour_utc", "block_events"}
missing = required_cols - set(df_block_hourly.columns)
if missing:
    raise RuntimeError(f"[Cell 27] Missing required columns in df_block_hour

# 5) Print summary (works for empty and non-empty)
total_block_events = int(df_block_hourly["block_events"].sum()) if not df_bl
hours_with_block_activity = int(len(df_block_hourly))

print("[Cell 27] A.18.2.3 — Hourly BLOCK activity prepared")
print(f"[Cell 27] hours_with_block_activity={hours_with_block_activity} tota

df_block_hourly.head(10)
```

```
[Cell 27] Using ISO evidence table: _
[Cell 27] rows=10 unique_ips=2
[Cell 27][INFO] No rows where expected_action == 'BLOCK' in this window.
[Cell 27] A.18.2.3 — Hourly BLOCK activity prepared
[Cell 27] hours_with_block_activity=0 total_block_events=0
```

Out[17]:

| event_hour_utc | block_events |
|---|---|

# Cell 28 — Audit Interpretation: Hourly BLOCK Evidence

**What this shows**

The hourly series quantifies how often the system identified `BLOCK`-eligible activity in each UTC hour of the observation window.

**Why it matters (ISO context)**

For ISO/IEC 27001:2022 **A.18.2.3**, a technical compliance review must be possible without relying on “live dashboards” or operator interpretation. This chart-ready dataset:

- enables periodic review (hourly buckets),
- can be compared across days/weeks (trend baselining),
- supports audit sampling (pick peak hours and inspect underlying events).

**Next**

The next cell will render a compliance plot from `df_block_hourly` so the PDF contains visual evidence, not only tables.

# Cell 29 — Distinguishing Persistent vs Burst Attack Patterns

**Objective**

This cell set classifies `BLOCK` -eligible activity into two operationally meaningful categories:

- **PERSISTENT**: repeated hostile behavior over extended time
- **BURST**: short-lived probing or scanning spikes

This distinction is essential for ISO/IEC 27001:2022 controls because it demonstrates that the organization does not treat all hostile activity equally, but instead applies **risk-aware analysis**.

**ISO Mapping**

- **A.12.6.1** — Management of technical vulnerabilities
- **A.16.1.7** — Evidence-based incident understanding
- **A.18.2.3** — Demonstrable technical review logic

The output supports audit review of *why* an IP is considered higher risk, not just *that* it appeared.

In [18]:
```
# Cell 30 — Classify BLOCK activity into PERSISTENT vs BURST patterns (audit

import pandas as pd

# Preconditions
if "df_block_hourly" not in globals():
    raise RuntimeError("[Cell 30] df_block_hourly not found. Run Cell 27 fir

# Re-discover ISO evidence table (defensive, explicit)
iso_df = None
REQUIRED = {"ts_utc", "ip_norm", "expected_action", "event_hour_utc"}

for name, obj in list(globals().items()):
    if isinstance(obj, pd.DataFrame) and REQUIRED.issubset(obj.columns):
        iso_df = obj
        iso_name = name
        break

if iso_df is None:
    raise RuntimeError("[Cell 30] ISO evidence DataFrame not found.")

print(f"[Cell 30] Using ISO evidence table: {iso_name}")

# Focus only on BLOCK-eligible events
df_block_events = iso_df[iso_df["expected_action"] == "BLOCK"].copy()
```

```
total_block = int(len(df_block_events))
unique_block_ips = int(df_block_events["ip_norm"].nunique())
print(f"[Cell 30] BLOCK events={total_block} unique_block_ips={unique_block_

if df_block_events.empty:
    print("[Cell 30][WARN] No BLOCK events in this window. Nothing to classi
    ip_persistence = pd.DataFrame(columns=[
        "ip_norm","block_events","first_seen_utc","last_seen_utc","persister
    ])
    display(ip_persistence)
else:
    # Aggregate per IP
    ip_persistence = (
        df_block_events
        .groupby("ip_norm", as_index=False)
        .agg(
            block_events=("ts_utc", "count"),
            first_seen_utc=("ts_utc", "min"),
            last_seen_utc=("ts_utc", "max"),
        )
    )

    # Compute persistence duration (hours)
    ip_persistence["persistence_hours"] = (
        (ip_persistence["last_seen_utc"] - ip_persistence["first_seen_utc"])
        .dt.total_seconds() / 3600.0
    )

    # Classification rule (simple, auditable, adjustable)
    # ≥ 1 hour duration OR ≥ 10 events → PERSISTENT
    PERSIST_HOURS = 1.0
    PERSIST_EVENTS = 10

    ip_persistence["attack_profile"] = ip_persistence.apply(
        lambda r: "PERSISTENT"
        if (r["persistence_hours"] >= PERSIST_HOURS or r["block_events"] >=
        else "BURST",
        axis=1,
    )


    # Summary counts (force stable ordering)
    profile_counts = ip_persistence["attack_profile"].value_counts().reindex
        ["PERSISTENT", "BURST"], fill_value=0
    )

    print("[Cell 30] Attack profile classification summary:")
    print(profile_counts)

    persistent_df = ip_persistence[ip_persistence["attack_profile"] == "PERS
        .sort_values(["block_events","persistence_hours"], ascending=[False,

    burst_df = ip_persistence[ip_persistence["attack_profile"] == "BURST"] \
        .sort_values(["block_events","persistence_hours"], ascending=[False,

    if len(persistent_df) == 0:
```

```
        print(f"\n[Cell 30] No PERSISTENT sources found under thresholds "
              f"(persistence_hours>={PERSIST_HOURS} OR block_events>={PERSIS
    else:
        print("\n[Cell 30] Top persistent threats:")
        display(persistent_df.head(10))

    print("\n[Cell 30] Top burst probes:")
    display(burst_df.head(10))

    # Show full table only if it adds information beyond what was already di
    if len(ip_persistence) > 10:
        print("\n[Cell 30] Full per-IP classification table:")
        display(ip_persistence)
```

```
[Cell 30] Using ISO evidence table: __
[Cell 30] BLOCK events=0 unique_block_ips=0
[Cell 30][WARN] No BLOCK events in this window. Nothing to classify.
```

| ip_norm | block_events | first_seen_utc | last_seen_utc | persistence_hours | attack_profile |
|---|---|---|---|---|---|

# Cell 31 — Audit Interpretation: Persistence-Based Threat Assessment

**What this demonstrates**

This classification shows that hostile activity is not treated as a single undifferentiated risk. Instead, the system distinguishes:

- **Persistent attackers** (sustained, high-risk, operational concern)
- **Burst probes** (short scans, lower long-term risk)

**Why auditors care**

ISO/IEC 27001 does not require blocking everything — it requires **reasoned, defensible decisions**. This table proves that:

- decisions can be justified retrospectively,
- escalation thresholds are explicit and reviewable,
- the same raw evidence can support different operational responses.

**Key point**

The logic is simple, transparent, and reproducible — exactly what auditors expect for technical compliance reviews.

# Cell 32 — ISO evidence enrichment (attribution + “who is attacking us?”)

**Goal (auditor-facing):** add external attribution (ASN / organization / country / network name) for the top BLOCK-eligible sources observed in `waf.events`.

This supports:

- **A.12.6.1** (technical vulnerability mgmt): identify recurring hostile infrastructure and concentration by provider/ASN.
- **A.16.1.7** (collection of evidence): preserve a reproducible attribution snapshot with timestamps.
- **A.18.2.3** (technical compliance review): demonstrate that policy outcomes map cleanly to observed evidence.

**Safety:** This section remains **read-only**. No ipset/iptables actions, no changes to enforcement state.

# Cell 33 — ISO evidence enrichment (attribution: ASN/org/country) with safe cache refresh

**Goal (auditor/accountant-focused):** Enrich the top BLOCK-eligible sources with external attribution (ASN, organization, country, network name) so we can show *who* is generating the hostile traffic.

**ISO evidence mapping:**

- **A.12.6.1** — identify recurring hostile infrastructure and concentration by provider/ASN
- **A.16.1.7** — preserve a reproducible attribution snapshot (timestamped, exportable)
- **A.18.2.3** — show evidence-to-policy traceability (BLOCK is explainable by source identity)

**Safety:** Read-only. No ipset/iptables execution. No enforcement state changes.

In [19]:
```
# ========================
# Cell 33 — Attribution enrichment for top BLOCK sources (self-contained; NO
# Fixes:
#  - No SystemExit (keeps notebook execution clean under nbconvert)
#  - Uses iso_events explicitly if present (deterministic)
#  - Handles expected_action values robustly (BLOCK/BLOCKED/RECOMMEND_BLOCK,
# ========================

import os, json, time
import pandas as pd
```

```python
# -------------------------
# 0) Locate ISO evidence table (deterministic preference order)
# -------------------------
REQUIRED = {"ts_utc", "ip_norm", "expected_action"}

iso_df = None
iso_name = None

# Prefer the canonical variable from Cell 5 if available
if "iso_events" in globals() and isinstance(globals()["iso_events"], pd.Data
    df0 = globals()["iso_events"]
    if REQUIRED.issubset(set(df0.columns)):
        iso_df = df0
        iso_name = "iso_events"

# Fallback: scan other dataframes safely
if iso_df is None:
    for name, obj in list(globals().items()):
        if isinstance(obj, pd.DataFrame) and REQUIRED.issubset(set(obj.colum
            iso_df = obj
            iso_name = name
            break

if iso_df is None:
    raise RuntimeError(
        "[Cell 33] ISO evidence DataFrame not found. Expected cols: "
        f"{sorted(REQUIRED)}. Run Cell 5 first."
    )

print(f"[Cell 33] Using ISO evidence table: {iso_name} rows={len(iso_df)}")

# -------------------------
# 1) Identify BLOCK-eligible rows robustly
# -------------------------
# Accept multiple representations across notebooks:
# - expected_action: BLOCK / BLOCKED / RECOMMEND_BLOCK
# - or "expected_action" might be "BLOCK" while "verdict" is SUSPICIOUS, etc
BLOCK_TOKENS = {"BLOCK", "BLOCKED", "RECOMMEND_BLOCK"}

ea = iso_df["expected_action"].astype(str).str.strip().str.upper()
df_block = iso_df[ea.isin(BLOCK_TOKENS)].copy()

# Optional fallback heuristic (only if nothing found):
# If expected_action doesn't contain BLOCK tokens, but verdict contains "SUS
# you can enable this via env var to avoid silently empty results.
if df_block.empty and os.environ.get("ISO_ALLOW_VERDICT_FALLBACK", "0").stri
    if "verdict" in iso_df.columns:
        vv = iso_df["verdict"].astype(str).str.strip().str.upper()
        df_block = iso_df[vv.eq("SUSPICIOUS")].copy()
        if not df_block.empty:
            print("[Cell 33][INFO] Fallback enabled: using verdict=='SUSPICI

TOPN = int(os.environ.get("WAF_RDAP_TOPN", "50"))

# Prepare empty outputs up-front (so downstream cells never break)
df_topN = pd.DataFrame(columns=["ip_norm", "block_events", "first_seen_utc",
```

```
df_iso_top_sources_enriched = pd.DataFrame(
    columns=[
        "ip_norm", "block_events", "first_seen_utc", "last_seen_utc", "persi
        "asn", "asn_org", "asn_country", "network_name", "network_country",
    ]
)

if df_block.empty:
    print("[Cell 33][WARN] No BLOCK-eligible events in this window (expectec
          f"{sorted(BLOCK_TOKENS)}).")
    print("[Cell 33] Producing empty df_topN / df_iso_top_sources_enriched (
    globals()["df_topN"] = df_topN
    globals()["df_iso_top_sources_enriched"] = df_iso_top_sources_enriched
    display(df_iso_top_sources_enriched)
else:
    # -------------------------
    # 2) Build df_topN deterministically from BLOCK-eligible events
    # -------------------------
    df_topN = (
        df_block.groupby("ip_norm", as_index=False)
        .agg(
            block_events=("ts_utc", "count"),
            first_seen_utc=("ts_utc", "min"),
            last_seen_utc=("ts_utc", "max"),
        )
    )

    df_topN["persistence_hours"] = (
        (df_topN["last_seen_utc"] - df_topN["first_seen_utc"]).dt.total_secc
    )

    # Sort: most events, then longest persistence
    df_topN = df_topN.sort_values(["block_events", "persistence_hours"], asc

    df_topN = df_topN.head(TOPN).reset_index(drop=True)

    print(f"[Cell 33] BLOCK sources: total_unique_ips={df_block['ip_norm'].r
    display(df_topN)

    globals()["df_topN"] = df_topN

    # -------------------------
    # 3) Cache helpers + RDAP lookup (ipwhois)
    # -------------------------
    CACHE_PATH = os.environ.get("WAF_GEO_CACHE", "/var/log/waf-snapshots/rda

    def load_cache(path: str) -> dict:
        try:
            with open(path, "r") as f:
                return json.load(f)
        except Exception:
            return {}

    def save_cache(path: str, cache: dict) -> None:
        tmp = path + ".tmp"
        os.makedirs(os.path.dirname(path), exist_ok=True)
```

```
        with open(tmp, "w") as f:
            json.dump(cache, f, indent=2, sort_keys=True)
        os.replace(tmp, path)

    def rdap_lookup_ipwhois(ip: str) -> dict:
        try:
            from ipwhois import IPWhois
            obj = IPWhois(ip)
            res = obj.lookup_rdap(depth=1)
            return {
                "ip_norm": ip,
                "asn": res.get("asn"),
                "asn_org": res.get("asn_description"),
                "asn_country": res.get("asn_country_code"),
                "network_name": (res.get("network") or {}).get("name"),
                "network_country": (res.get("network") or {}).get("country")
            }
        except Exception as e:
            return {"_error": str(e)}

    def cache_missing_asn(item: dict) -> bool:
        if not isinstance(item, dict):
            return True
        asn = item.get("asn")
        return (asn is None) or (str(asn).strip() in {"", "NA", "0", "NONE"}

    # -------------------------
    # 4) Enrichment with "poisoned cache" fix: re-lookup if ASN missing
    # -------------------------
    ips = df_topN["ip_norm"].astype(str).tolist()
    cache = load_cache(CACHE_PATH)

    todo = []
    for ip in ips:
        if ip not in cache:
            todo.append(ip)
        else:
            if cache_missing_asn(cache.get(ip, {})):
                todo.append(ip)

    MAX_LOOKUPS = int(os.environ.get("WAF_RDAP_MAX", "200"))
    todo = todo[:MAX_LOOKUPS]

    print(f"[Cell 33][RDAP] Will attempt RDAP (ipwhois) for {len(todo)} IPs

    new_ok = 0
    new_fail = 0
    for ip in todo:
        res = rdap_lookup_ipwhois(ip)
        if "_error" in res:
            new_fail += 1
            print(f"[Cell 33][RDAP][WARN] RDAP failed for {ip}: {res['_error
            # Do NOT overwrite good cache entries with failures
            continue
        cache[ip] = res
        new_ok += 1
```

```
        time.sleep(0.05)

    save_cache(CACHE_PATH, cache)
    print(f"[Cell 33][RDAP] New successful lookups: {new_ok}; failures: {new

    # Build attribution df from cache
    rows = []
    for ip in ips:
        item = cache.get(ip, {})
        rows.append({
            "ip_norm": ip,
            "asn": item.get("asn"),
            "asn_org": item.get("asn_org"),
            "asn_country": item.get("asn_country"),
            "network_name": item.get("network_name"),
            "network_country": item.get("network_country"),
        })

    df_attr = pd.DataFrame(rows)
    df_iso_top_sources_enriched = df_topN.merge(df_attr, on="ip_norm", how="

    print(f"[Cell 33] Enriched top sources table rows={len(df_iso_top_source
    display(df_iso_top_sources_enriched)

    globals()["df_iso_top_sources_enriched"] = df_iso_top_sources_enriched
```

```
[Cell 33] Using ISO evidence table: iso_events rows=10166
[Cell 33][WARN] No BLOCK-eligible events in this window (expected_action not
in ['BLOCK', 'BLOCKED', 'RECOMMEND_BLOCK']).
[Cell 33] Producing empty df_topN / df_iso_top_sources_enriched (no failur
e).
```

| ip_norm | block_events | first_seen_utc | last_seen_utc | persistence_hours | asn | asn_org |
|---|---|---|---|---|---|---|

# Cell 34 — ISO 27001 Control Evidence Rollup (A.12.6.1 / A.16.1.7 / A.18.2.3)

**Goal (auditor/accountant-focused):** Produce a concise, exportable table that maps the *database-backed event corpus* and our *classification outcomes* to three ISO/IEC 27001:2022 Annex A evidence areas:

- **A.12.6.1** — demonstrate recurring hostile infrastructure and concentration (IP/ASN), supporting vulnerability/threat management.
- **A.16.1.7** — demonstrate that evidence is collected, preserved, and reproducible (timestamped events, windowed queries, rollups).
- **A.18.2.3** — demonstrate technical compliance review by showing alignment between `verdict` and `expected_action` outcomes.

This cell writes CSV artifacts to `/var/log/waf-snapshots/` for audit trail retention and offline review.

In [20]:
```
# Cell 34 — ISO control evidence rollup + export (adds attribution coverage

import os
import pandas as pd
from datetime import datetime, timezone

# Preconditions
if "df_iso_top_sources_enriched" not in globals():
    raise RuntimeError("[Cell 34] df_iso_top_sources_enriched not found. Rur

df_top_enriched = df_iso_top_sources_enriched

# Discover ISO evidence df (defensive)
REQUIRED = {"ts_utc", "ip_norm", "verdict", "expected_action", "event_hour_u
iso_df = None
iso_name = None

for name, obj in list(globals().items()):
    if isinstance(obj, pd.DataFrame) and REQUIRED.issubset(set(obj.columns))
        iso_df = obj
        iso_name = name
        break

if iso_df is None:
    raise RuntimeError("[Cell 34] ISO evidence DataFrame not found.")

print(f"[Cell 34] Using ISO evidence table: {iso_name}")

# Window metrics (from ISO evidence table)
now_utc = datetime.now(timezone.utc)
stamp = now_utc.strftime("%Y%m%d_%H%M%S")
day = now_utc.strftime("%Y%m%d")
```

```
total_events = int(len(iso_df))
unique_ips = int(iso_df["ip_norm"].nunique())
hours_observed = int(iso_df["event_hour_utc"].nunique())

block_events = int((iso_df["expected_action"] == "BLOCK").sum())
trace_events = int((iso_df["expected_action"] == "TRACE").sum())
other_events = int((iso_df["expected_action"] == "OTHER").sum())

# Attribution coverage (from enriched top sources)
top_rows = int(len(df_top_enriched))
top_unique_ips = int(df_top_enriched["ip_norm"].nunique()) if "ip_norm" in c

asn_known = 0
asn_coverage = 0.0
asn_unique = 0
if "asn" in df_top_enriched.columns:
    asn_known = int(df_top_enriched["asn"].notna().sum())
    asn_coverage = (asn_known / top_rows) if top_rows > 0 else 0.0
    asn_unique = int(df_top_enriched.loc[df_top_enriched["asn"].notna(), "as

df_iso_controls = pd.DataFrame([
    {
        "iso_clause": "A.12.6.1",
        "title": "Management of technical vulnerabilities",
        "evidence_focus": "Recurring hostile infrastructure by IP/ASN; conce
        "key_metrics": (
            f"BLOCK_events={block_events}, unique_ips={unique_ips}, hours_ob
            f"top_rows={top_rows} (top_unique_ips={top_unique_ips}), "
            f"asn_coverage={asn_known}/{top_rows}={asn_coverage:.2%}, asn_ur
        ),
        "artifacts": "Independent notebook (DB-backed) + enriched top source
    },
    {
        "iso_clause": "A.16.1.7",
        "title": "Collection of evidence",
        "evidence_focus": "Timestamped, queryable event corpus with reproduc
        "key_metrics": f"total_events={total_events}, hours_observed={hours_
        "artifacts": "waf.events + hourly rollups + daily snapshots (CSV) +
    },
    {
        "iso_clause": "A.18.2.3",
        "title": "Technical compliance review",
        "evidence_focus": "Verdict vs expected_action alignment; policy outc
        "key_metrics": f"BLOCK={block_events}, TRACE={trace_events}, OTHER={
        "artifacts": "Expected_action rollups + compliance alignment tables
    },
])

print("\n[Cell 34] ISO control evidence rollup:")
display(df_iso_controls)

OUTDIR = os.environ.get("WAF_SNAP_DIR", "/var/log/waf-snapshots")
os.makedirs(OUTDIR, exist_ok=True)

controls_path = f"{OUTDIR}/iso_controls_{day}.csv"
```

```
controls_path_ts = f"{OUTDIR}/iso_controls_{stamp}.csv"
tops_path = f"{OUTDIR}/iso_top_sources_enriched_{day}.csv"
tops_path_ts = f"{OUTDIR}/iso_top_sources_enriched_{stamp}.csv"

df_iso_controls.to_csv(controls_path, index=False)
df_iso_controls.to_csv(controls_path_ts, index=False)
df_top_enriched.to_csv(tops_path, index=False)
df_top_enriched.to_csv(tops_path_ts, index=False)

print(f"[Cell 34] Wrote: {controls_path}")
print(f"[Cell 34] Wrote: {controls_path_ts}")
print(f"[Cell 34] Wrote: {tops_path}")
print(f"[Cell 34] Wrote: {tops_path_ts}")

print(f"[Cell 34] Attribution coverage: asn_known={asn_known}/{top_rows} ({a
```

```
[Cell 34] Using ISO evidence table: __

[Cell 34] ISO control evidence rollup:
```

| | iso_clause | title | evidence_focus | key_metrics | artifacts |
|---|---|---|---|---|---|
| **0** | A.12.6.1 | Management of technical vulnerabilities | Recurring hostile infrastructure by IP/ASN; co... | BLOCK_events=0, unique_ips=2, hours_observed=1... | Independent notebook (DB-backed) + enriched to... |
| **1** | A.16.1.7 | Collection of evidence | Timestamped, queryable event corpus with repro... | total_events=10, hours_observed=1, unique_ips=2 | waf.events + hourly rollups + daily snapshots ... |
| **2** | A.18.2.3 | Technical compliance review | Verdict vs expected_action alignment; policy o... | BLOCK=0, TRACE=0, OTHER=0 | Expected_action rollups + compliance alignment... |

```
[Cell 34] Wrote: /var/log/waf-snapshots/iso_controls_20260301.csv
[Cell 34] Wrote: /var/log/waf-snapshots/iso_controls_20260301_214503.csv
[Cell 34] Wrote: /var/log/waf-snapshots/iso_top_sources_enriched_20260301.cs
v
[Cell 34] Wrote: /var/log/waf-snapshots/iso_top_sources_enriched_20260301_21
4503.csv
[Cell 34] Attribution coverage: asn_known=0/0 (0.00%), asn_unique=0
```

In [ ]:

In [ ]:

In [ ]: