Observation API Integration Guide

This guide walks through integrating with the Observation Event Monitoring API for continuous event collection. The API is designed for SIEM platforms (Sentinel, Splunk, Cribl) and any system that needs to continuously ingest events.

Architecture: Why Cursor-Based Streaming

The API uses a **cursor-based streaming model** instead of a date-range query model. Understanding this distinction is critical to a successful integration.

Cursor-Based Streaming (This API)

```
Your System                          Observation API
-----------                          ---------------
Store cursor = 0
    |
    +--> GET /v1/events?cursor=0 --> Returns events 1-1000
    |                                X-Next-Cursor: 1000
    |                                X-Has-More: true
Store cursor = 1000
    |
    +--> GET /v1/events?cursor=1000 --> Returns events 1001-1500
    |                                   X-Next-Cursor: 1500
    |                                   X-Has-More: false
Store cursor = 1500
    |
    ... wait for polling interval ...
    |
    +--> GET /v1/events?cursor=1500 --> Returns events 1501-1520
    |                                   X-Next-Cursor: 1520
    |                                   X-Has-More: false
Store cursor = 1520
    |
    ... repeat forever ...
```

What you manage: One integer (the cursor).

What the API guarantees:

- No missed events: every event with an ID greater than your cursor is returned.
- No duplicates: advancing the cursor excludes previously seen events.
- Chronological order: events are returned oldest-first.

Why Not Date-Range Queries?

A date-range query API (e.g., `?earliest=2025-01-15T00:00:00&latest=2025-01-15T23:59:59`) requires your integration to:

1. Track the last timestamp you queried
2. Handle overlapping time windows (events arriving with past timestamps)
3. Implement separate paging within each time window
4. Manage gap detection between polling intervals
5. Handle clock skew between your system and the server

The cursor eliminates all of this. **The cursor IS your bookmark.** It combines paging and time progression into a single value.

Quick Start

Step 1: Make Your First Request

```
curl -s -D - \
  -H "Authorization: Bearer obs_YOUR_API_KEY_HERE" \
  "https://your-domain.com/v1/events"
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/x-ndjson
X-Next-Cursor: 42
X-Has-More: true

{"event_id":"ts-a1b2c3d4-...","type":"transfer_sent","timestamp":"2025-01-15T10:30:00","data":{...}}
{"event_id":"fd-b2c3d4e5-...","type":"file_downloaded","timestamp":"2025-01-15T10:31:00","data":{...}}
```

Step 2: Save the Cursor

Store the value of the `X-Next-Cursor` header. This is the only state you need to persist between calls.

Step 3: Request the Next Page

```
curl -s -D - \
  -H "Authorization: Bearer obs_YOUR_API_KEY_HERE" \
  "https://your-domain.com/v1/events?cursor=42"
```

Step 4: Repeat Until Caught Up

Continue calling with the latest cursor until `X-Has-More` is `false`. Then wait for your polling interval and repeat.

Reference Implementation (Python)

```
import time
import requests

API_URL = "https://your-domain.com/v1/events"
API_KEY = "obs_YOUR_API_KEY_HERE"
POLL_INTERVAL_SECONDS = 90  # Must exceed the 60-second rate limit

def collect_events():
    """Continuously collect events using cursor-based pagination."""
    cursor = load_cursor()  # Load from persistent storage, or 0 if first run

    while True:
        # Drain all available pages
        while True:
            events, cursor, has_more = fetch_page(cursor)

            if events:
                process_events(events)
                save_cursor(cursor)  # Persist after each successful page

            if not has_more:
                break

        # All caught up -- wait before polling again
        time.sleep(POLL_INTERVAL_SECONDS)
```

```python
def fetch_page(cursor):
    """Fetch one page of events. Returns (events, next_cursor, has_more)."""
    params = {"limit": 1000}
    if cursor > 0:
        params["cursor"] = cursor

    response = requests.get(
        API_URL,
        headers={"Authorization": f"Bearer {API_KEY}"},
        params=params,
        timeout=30,
    )

    if response.status_code == 429:
        retry_after = int(response.headers.get("Retry-After", 60))
        print(f"Rate limited. Waiting {retry_after} seconds.")
        time.sleep(retry_after)
        return [], cursor, True  # Retry the same cursor

    response.raise_for_status()

    next_cursor = int(response.headers.get("X-Next-Cursor", cursor))
    has_more = response.headers.get("X-Has-More", "false") == "true"

    events = []
    for line in response.text.strip().split("\n"):
        if line:
            events.append(json.loads(line))

    return events, next_cursor, has_more


def process_events(events):
    """Send events to your SIEM or processing pipeline."""
    for event in events:
        # Forward to Sentinel, Splunk, or your log pipeline
        print(f"  {event['event_id']} | {event['type']} | {event['timestamp']}")


def load_cursor():
    """Load the last cursor from persistent storage."""
    try:
        with open("observation_cursor.txt", "r") as f:
            return int(f.read().strip())
    except (FileNotFoundError, ValueError):
        return 0


def save_cursor(cursor):
    """Persist cursor so we resume correctly after restart."""
    with open("observation_cursor.txt", "w") as f:
        f.write(str(cursor))


if __name__ == "__main__":
    import json
    collect_events()
```

Reference Implementation (Bash / curl)

For simple integrations or testing:

```bash
#!/bin/bash
set -e
```

```bash
API_URL="https://your-domain.com/v1/events"
API_KEY="obs_YOUR_API_KEY_HERE"
CURSOR_FILE="observation_cursor.txt"
POLL_INTERVAL=90

# Load or initialize cursor
if [ -f "$CURSOR_FILE" ]; then
    CURSOR=$(cat "$CURSOR_FILE")
else
    CURSOR=0
fi

echo "Starting event collection from cursor $CURSOR"

while true; do
    # Drain all available pages
    HAS_MORE="true"
    while [ "$HAS_MORE" = "true" ]; do
        RESPONSE=$(curl -s -D /tmp/obs_headers.txt \
            -H "Authorization: Bearer $API_KEY" \
            "$API_URL?cursor=$CURSOR&limit=1000" \
            -w "\n%{http_code}")

        HTTP_CODE=$(echo "$RESPONSE" | tail -1)
        BODY=$(echo "$RESPONSE" | sed '$d')

        if [ "$HTTP_CODE" = "429" ]; then
            RETRY=$(grep -i "Retry-After" /tmp/obs_headers.txt | tr -d '[:space:]' | cut -d: -f2)
            echo "Rate limited. Waiting ${RETRY:-60} seconds."
            sleep "${RETRY:-60}"
            continue
        fi

        if [ "$HTTP_CODE" != "200" ]; then
            echo "Error: HTTP $HTTP_CODE"
            echo "$BODY"
            sleep 30
            break
        fi

        # Extract cursor and has_more from response headers
        CURSOR=$(grep -i "X-Next-Cursor" /tmp/obs_headers.txt | tr -d '[:space:]' | cut -d: -f2)
        HAS_MORE=$(grep -i "X-Has-More" /tmp/obs_headers.txt | tr -d '[:space:]' | cut -d: -f2)

        # Process events (pipe to your SIEM ingestion)
        if [ -n "$BODY" ]; then
            echo "$BODY"
            # Example: echo "$BODY" | your-siem-forwarder --input ndjson
        fi

        # Persist cursor
        echo "$CURSOR" > "$CURSOR_FILE"
    done

    echo "Caught up at cursor $CURSOR. Sleeping ${POLL_INTERVAL}s..."
    sleep $POLL_INTERVAL
done
```

Integration Patterns

Pattern 1: Continuous Polling (Recommended)

Best for real-time monitoring. Run as a service or scheduled task.

```
Loop:
  1. Drain all pages (cursor -> cursor -> cursor until has_more=false)
```

```
  2. Sleep for polling interval (e.g., 90 seconds)
  3. Repeat from step 1
```

Polling interval: Must be greater than the rate limit interval (60 seconds). Recommended: 90 seconds.

Pattern 2: Scheduled Batch Collection

Best for periodic ingestion (e.g., every 5 minutes via cron).

```
Cron job (every 5 minutes):
  1. Load cursor from persistent storage
  2. Drain all pages
  3. Save final cursor
  4. Exit
```

Note: With a 60-second rate limit, a 5-minute cron window allows up to 5 pages per run (5,000 events). If your event volume regularly exceeds this, use continuous polling instead.

Pattern 3: Initial Backfill + Continuous

For new integrations that need to ingest historical events:

```
Phase 1 (backfill):
  1. Start with cursor=0
  2. Drain ALL pages (respecting rate limits)
  3. This may take time for large event histories

Phase 2 (continuous):
  1. Continue with the cursor from phase 1
  2. Switch to normal polling interval
```

The same code handles both phases -- no separate backfill logic is needed.

Handling the 1,000-Event Limit

The `limit` parameter caps events **per page**, not total. To collect all available events:

```
Call 1:  GET /v1/events?cursor=0&limit=1000
         -> 1000 events, X-Next-Cursor: 1000, X-Has-More: true

Call 2:  GET /v1/events?cursor=1000&limit=1000
         -> 1000 events, X-Next-Cursor: 2000, X-Has-More: true

Call 3:  GET /v1/events?cursor=2000&limit=1000
         -> 500 events, X-Next-Cursor: 2500, X-Has-More: false

Total collected: 2500 events across 3 pages.
```

Each call is subject to the rate limit. With a 60-second interval, draining 5,000 events takes approximately 5 minutes (5 pages x 60 seconds).

Sentinel Integration Notes

When configuring a Sentinel data connector:

1. **Data connector type:** Use a custom REST API connector or Azure Function.
2. **Authentication:** Bearer token (set in connector configuration).
3. **Polling:** Configure the polling interval to at least 90 seconds.

4. **State management:** Store the `X-Next-Cursor` value between polling cycles. Most Sentinel connectors support a checkpoint or state file mechanism for this.
5. **Response format:** Parse NDJSON (one JSON object per line). Each line is a complete, independent JSON object.
6. **Rate limit handling:** Implement retry logic using the `Retry-After` header on 429 responses.

Splunk Integration Notes

When configuring a Splunk HTTP Event Collector (HEC) integration:

1. **Scripted input:** Use a scripted input or modular input that calls the API.
2. **Checkpoint:** Store the cursor in a Splunk checkpoint file (`$SPLUNK_HOME/var/lib/splunk/modinputs/`).
3. **Source type:** Set sourcetype to `observation:events` or similar.
4. **The cursor replaces `earliest_time` / `latest_time` :** You do not need to configure time-based filtering. The cursor handles this automatically.
5. **collect command limit:** The 1,000-event limit per page is unrelated to Splunk's `collect` command. Page through using the cursor to retrieve all events.

Troubleshooting

"I'm hitting the 1,000 limit and not getting all events"

The 1,000 limit is per page. Check the `X-Has-More` header -- if it is `"true"` , call again with the `X-Next-Cursor` value to get the next page. Repeat until `X-Has-More` is `"false"` .

"I'm getting 429 Too Many Requests"

You are calling the API more than once per 60 seconds. Read the `Retry-After` header and wait that many seconds before retrying. Set your polling interval to at least 90 seconds.

"I need to query events from a specific date range"

The API does not support date-range queries directly. Instead:

- **For new integrations:** Start with `cursor=0` to ingest all historical events. The cursor advances chronologically, so you will process events in time order.
- **For re-ingestion:** Reset your stored cursor to `0` and re-process from the beginning. Filter by timestamp on your end if you only need a specific window.

The cursor-based design ensures no missed events and no duplicates, which is more reliable than date-range queries for continuous monitoring.

"What happens if my integration goes down for a while?"

Nothing is lost. When your integration restarts, it resumes from the last saved cursor and picks up all events that accumulated while it was down. There is no expiration on the cursor position.

"Can multiple consumers use the same API key?"

Yes, but each consumer must maintain its own cursor independently. The server does not track cursor state -- it is entirely client-side. However, all consumers sharing a key share the same rate limit.