Chronicle-Sniffer

A Scalable Wireshark-to-SecOps Pipeline on GCP

Filippo Lucchesi
filippo.lucchesi@studio.unibo.it

Scalable and Reliable Services M
Università di Bologna

10 Giugno 2025

Indice

1 Introduzione

Questo report descrive l'architettura, l'implementazione e le caratteristiche di un progetto sviluppato per il corso di "Scalable and Reliable Services M" presso l'Università di Bologna.

Il sistema, denominato "Chronicle-Sniffer", realizza una pipeline di elaborazione dati di rete su Google Cloud Platform (GCP), con l'obiettivo primario di catturare pacchetti di rete (PCAP e PCAPNG), processarli in modo scalabile e affidabile, e trasformarli nel formato Unified Data Model (UDM) per successive analisi, tipicamente in contesti di sicurezza informatica, er esempio avvalendosi del Software Google SecOps.

Il progetto rappresenta una significativa evoluzione di un precedente lavoro concepito per il corso di Cybersecurity (Wireshark-to-Chronicle-Pipeline). Mentre la versione iniziale si concentrava sulla logica di conversione locale, questa iterazione si focalizza sulla creazione di un'infrastruttura cloud-nativa robusta, enfatizzando scalabilità, resilienza, gestibilità e osservabilità. Le tecnologie chiave impiegate includono servizi gestiti di GCP come Cloud Storage, Pub/Sub (con supporto per Dead-Letter Queue - DLQ), e Cloud Run per l'elaborazione serverless. L'intera infrastruttura è definita e gestita tramite Terraform, promuovendo l'approccio Infrastructure as Code (IaC), e include la definizione di una dashboard operativa personalizzata in Cloud Monitoring. I componenti software sono containerizzati con Docker. L'analisi del traffico di rete si avvale di TShark per la cattura e la conversione iniziale, mentre lo script Python `json2udm_cloud.py` è stato reingegnerizzato per un parsing JSON efficiente in streaming e una mappatura UDM più dettagliata.

L'obiettivo è dimostrare la capacità di costruire un servizio distribuito che non solo risponda a requisiti funzionali specifici, ma che sia anche progettato per operare in maniera efficiente, robusta, sicura e altamente osservabile in un ambiente cloud.

2 Architettura Generale

Il sistema Chronicle-Sniffer è progettato come una pipeline *event-driven*, che sfrutta diversi servizi GCP per orchestrare il flusso dei dati dalla cattura del traffico di rete fino alla sua trasformazione finale. L'architettura si articola attorno a componenti principali che interagiscono in modo asincrono, garantendo disaccoppiamento e resilienza.

- **Sniffer (On-Premises/Edge):** Un container Docker, eseguito su una macchina fisica o virtuale nell'ambiente da monitorare. Utilizza `tshark` per catturare il traffico di rete (supportando formati `.pcap` e `.pcapng`), gestisce la rotazione automatica dei file di cattura e, una volta completato un file, lo carica su un bucket Google Cloud Storage (GCS) (denominato `incoming-pcaps`). Immediatamente dopo, pubblica una notifica contenente il nome del file su un topic Google Cloud Pub/Sub. Lo sniffer include anche una funzionalità di *heartbeat* per monitorarne lo stato.
- **Google Cloud Pub/Sub:** Agisce come *message broker*. Un topic principale riceve le notifiche dallo sniffer. Una sottoscrizione *push*, configurata con autenticazione OIDC sicura, inoltra i messaggi all'endpoint del servizio Cloud Run. È previsto un topic Dead-Letter Queue (DLQ) per gestire i messaggi che non possono essere processati correttamente dopo vari tentativi.
- **Google Cloud Storage (GCS):** Due bucket distinti sono utilizzati:
  - `incoming-pcaps`: Funge da area di *staging* per i file di cattura raw. Come visibile nello screenshot seguente (Figura 2), questo bucket riceve i file `.pcap` direttamente dallo sniffer.

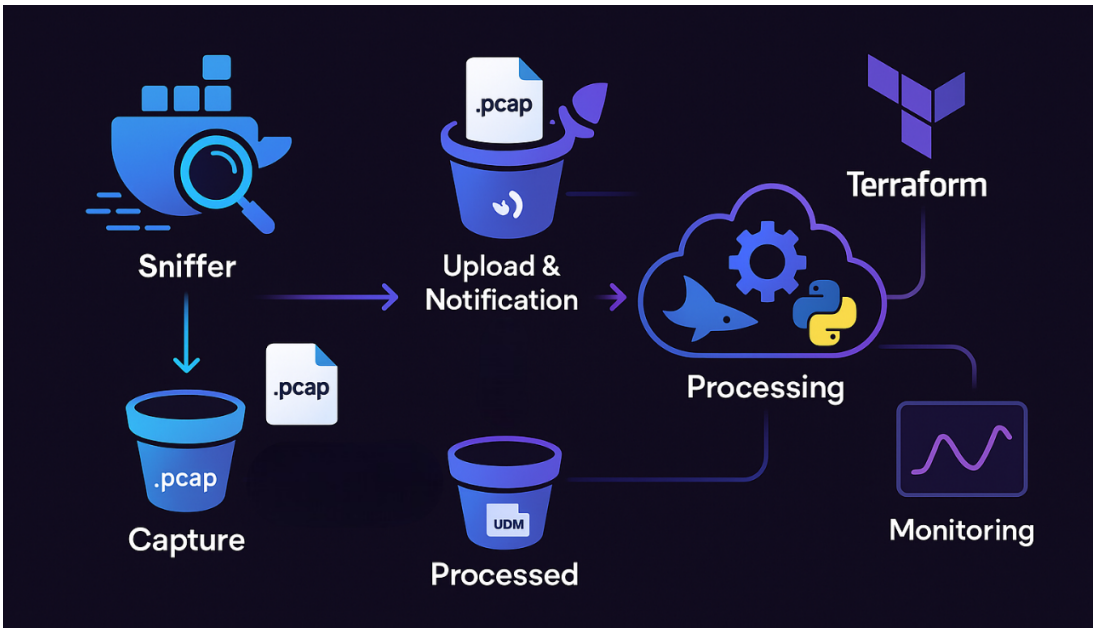

Figura 1: Diagramma dell'architettura generale del sistema Chronicle-Sniffer.

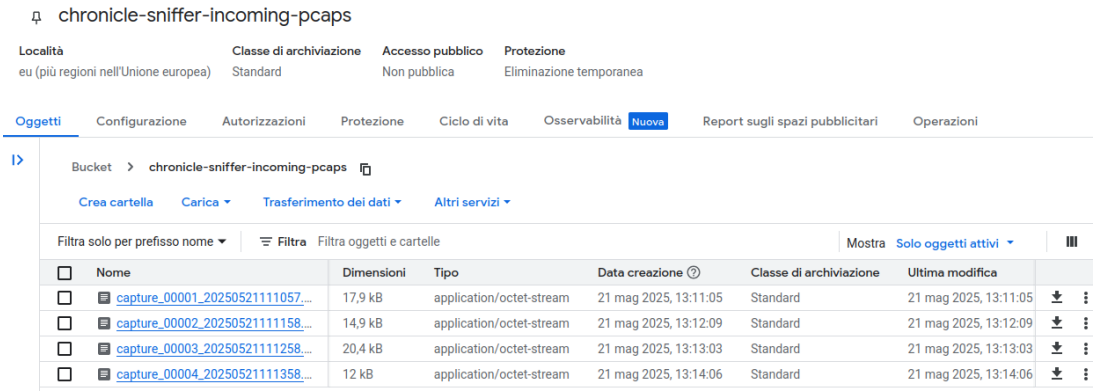

Figura 2: Contenuto del bucket GCS `chronicle-sniffer-incoming-pcaps`.

– `processed-udm`: Archivia i file JSON finali contenenti gli eventi di rete trasformati nel formato UDM. Lo screenshot successivo (Figura 3) mostra un esempio dei file `.udm.json` prodotti dal Processor.

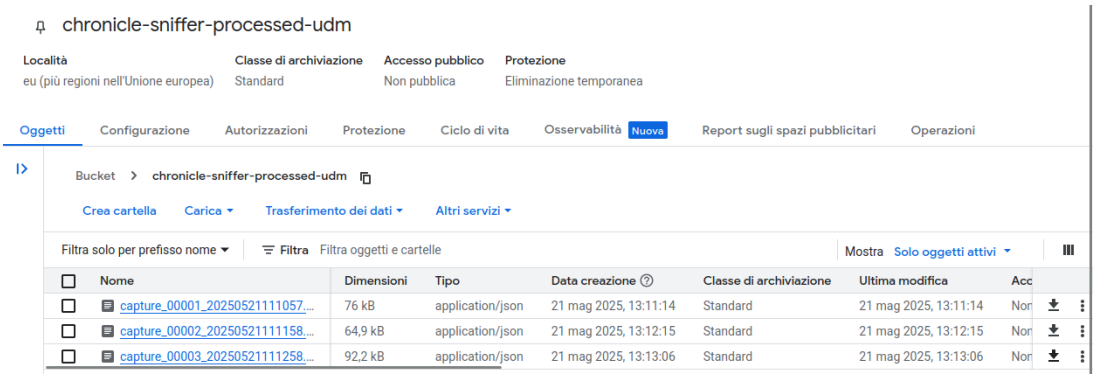

Figura 3: Contenuto del bucket GCS `chronicle-sniffer-processed-udm`.

- **Processore (Google Cloud Run):** Un servizio serverless che esegue un container Docker. Attivato dalle notifiche Pub/Sub, il servizio:
  1. Scarica il file di cattura specificato da GCS.
  2. Utilizza `tshark` (installato nel container) per convertire il file di cattura in un formato JSON strutturato.
  3. Esegue lo script Python `json2udm_cloud.py`, che ora utilizza la libreria `ijson` per il parsing in streaming del (potenzialmente grande) output JSON di `tshark`. Questo approccio è cruciale per l'efficienza della memoria in Cloud Run. Lo script mappa ogni pacchetto nel formato UDM.
  4. Carica il file UDM JSON risultante nel bucket GCS `processed-udm`.

  Il servizio è progettato per scalare automaticamente e include *health probes* per il monitoraggio da parte di Cloud Run.

L'intera infrastruttura, inclusi i bucket, il topic Pub/Sub con DLQ, il servizio Cloud Run, i Service Account, i permessi IAM, le metriche basate su log e una dashboard operativa personalizzata, è definita e gestita tramite Terraform.

2.1 Motivazioni delle Scelte Architetturali

L'architettura è stata definita per massimizzare scalabilità, affidabilità, gestibilità, efficienza dei costi e sicurezza:

- **Scalabilità:** Cloud Run e Pub/Sub scalano automaticamente per gestire carichi variabili. L'elaborazione in streaming nel Processor previene colli di bottiglia legati all'esaurimento della memoria.
- **Affidabilità e Resilienza:** Il disaccoppiamento tramite Pub/Sub, unito al supporto DLQ, garantisce che i messaggi non vengano persi e che i fallimenti siano gestiti in modo isolato. Cloud Run gestisce la disponibilità delle istanze del processor.
- **Gestibilità (IaC):** Terraform permette una gestione dichiarativa, versionabile e automatizzata dell'intera infrastruttura, inclusa la dashboard di monitoraggio.
- **Efficienza dei costi:** I servizi serverless e lo scaling a zero di Cloud Run ottimizzano i costi.
- **Sicurezza:** Adozione del principio del *least privilege* per i Service Account, autenticazione OIDC tra Pub/Sub e Cloud Run, e gestione sicura delle chiavi per lo sniffer on-premises.
- **Osservabilità:** Integrazione profonda con Cloud Logging e Cloud Monitoring, con metriche personalizzate basate su log (LBM) e una dashboard operativa dedicata per una visione completa dello stato e delle prestazioni della pipeline.

3 Logica di Conversione e Adattamenti per il Cloud

Il cuore analitico del sistema è lo script Python `json2udm_cloud.py`, eseguito dal componente Processor. Questo script è responsabile della trasformazione dei dati dei pacchetti, precedentemente convertiti in JSON da `tshark`, nel formato standardizzato Unified Data Model (UDM). È importante sottolineare che la logica dettagliata di parsing dei singoli campi PCAP e la loro mappatura iniziale ai concetti UDM sono state approfondite nel precedente progetto di Cybersecurity. Il focus di questo progetto è stato l'adattamento di tale logica per un ambiente cloud efficiente e la costruzione di un'infrastruttura scalabile e resiliente attorno ad essa.

3.1 Evoluzione per l'Efficienza Cloud: da `json.loads` a `ijson`

Nel progetto di Cybersecurity, la conversione JSON → UDM avveniva caricando l'intero output JSON di `tshark` in memoria tramite `json.loads()`. Sebbene funzionale per file di dimensioni moderate, questo approccio diventa proibitivo in un ambiente serverless come Cloud Run quando si trattano file PCAP di grandi dimensioni, che possono generare output JSON di svariati gigabyte, portando a errori di Out of Memory (OOM).

La modifica più significativa in `json2udm_cloud.py` è l'adozione della libreria `ijson`. Questa libreria permette il parsing JSON in streaming: invece di caricare l'intero documento, `ijson` processa il file JSON in modo incrementale, leggendo e interpretando gli oggetti JSON (in questo caso, i singoli pacchetti) uno alla volta.

```
# Lo script apre il file JSON prodotto da tshark in modalita binaria ('rb').
# ijson.items(f_json, 'item') crea un iteratore che assume che il file JSON
# sia un array di oggetti alla radice. 'item' istruisce ijson a restituire
# (yield) ciascun elemento di questo array principale, uno alla volta.
# Questo permette di processare pacchetti individualmente senza caricare
# l'intero array di pacchetti (potenzialmente enorme) in memoria.
try:
    with open(json_file_path, 'rb') as f_json: # Modalita binaria per ijson
        json_packet_iterator = ijson.items(f_json, 'item')
        for packet_data_dict in json_packet_iterator:
            # Ogni 'packet_data_dict' e' un dizionario Python
            # rappresentante un singolo pacchetto.
            udm_event = convert_single_packet_to_udm(packet_data_dict)
            udm_events_list.append(udm_event)
            processed_packet_count += 1
            # ... (logica per contare errori specifici per pacchetto)
except ijson.JSONError as e_ijson:
    # Gestisce errori se il file JSON non e' ben formato
    # o non e' un array alla radice.
    logging.error(f"ijson.JSONError while parsing streaming JSON from {json_file_path}: {
    e_ijson}...")
    return []
```

Listing 1: Estratto da `json2udm_cloud.py`: Utilizzo di `ijson` per lo streaming.

Questo approccio riduce drasticamente il picco di utilizzo della memoria, rendendo il processo di conversione adatto anche per istanze Cloud Run con risorse limitate e capace di gestire file di input molto grandi.

3.2 Funzionamento Adattato dello script `json2udm_cloud.py`

Oltre al parsing in streaming, lo script `json2udm_cloud.py` è stato migliorato per:

- **Robustezza:** Ogni pacchetto in input produce un evento UDM, anche se si tratta di un evento minimo che segnala un errore di elaborazione per quel pacchetto. La conversione dei timestamp (`convert_timestamp_robust`) è più resiliente, con fallback all'ora corrente in caso di formati imprevisti, garantendo la conformità UDM.

```
# Questa funzione tenta diverse strategie per parsare il timestamp fornito da
    tshark.
# Inizia con il formato piu comune (con microsecondi), poi tenta formati
    alternativi
# (es. senza microsecondi, pulendo stringhe di fuso orario comuni).
# Se tutte le conversioni falliscono, logga un warning e restituisce l'ora corrente
# in formato ISO 8601 UTC, assicurando che il campo 'event_timestamp' dell'UDM
# sia sempre presente e valido.
def convert_timestamp_robust(timestamp_str):
    if not timestamp_str:
        logging.warning(f"Timestamp string is missing or empty. Using current time
    as fallback.")
        return datetime.now(timezone.utc).isoformat(timespec='microseconds').
    replace('+00:00', 'Z')
    try:
```

```
        # Tentativo primario: formato standard di tshark con microsecondi
        dt_naive = datetime.strptime(timestamp_str[:26], "%b %d, %Y %H:%M:%S.%f")
    except ValueError:
        # Fallback: prova a pulire stringhe di fuso orario e a parsare senza microsecondi
        try:
            cleaned_ts = timestamp_str.split(" UTC")[0]
            # Altri tentativi di pulizia potrebbero essere aggiunti qui
            cleaned_ts = cleaned_ts.split(" Central European Summer Time")[0].strip()
            dt_naive = datetime.strptime(cleaned_ts, "%b %d, %Y %H:%M:%S")
        except ValueError as e_fallback: # e_fallback non usato, rimosso per pulizia warning
            logging.warning(f"Error converting timestamp '{timestamp_str}' ... Using current time.")
            return datetime.now(timezone.utc).isoformat(timespec='microseconds').replace('+00:00', 'Z')

    # Assicura che il datetime sia 'aware' (consapevole del fuso orario) e in UTC.
    dt_aware = dt_naive.replace(tzinfo=timezone.utc)
    # Formatta in ISO 8601, assicurando la 'Z' per Zulu time (UTC).
    iso_timestamp = dt_aware.isoformat(timespec='microseconds').replace('+00:00', 'Z')
    return iso_timestamp

```

Listing 2: Estratto da `json2udm_cloud.py`: Funzione `convert_timestamp_robust`.

- **Struttura UDM Allineata:** La struttura degli eventi UDM generati è stata raffinata per aderire più strettamente alle aspettative di sistemi come Chronicle, con sezioni `metadata`, `principal`, `target`, `network` (che può contenere `application_protocol_data` per HTTP, DNS, TLS, annidato sotto `network`) e `additional` ben definite.

```
# ... (all'interno di convert_single_packet_to_udm)
udm_payload = {
    "metadata": {
        "event_timestamp": event_timestamp,
        "product_name": "Wireshark TShark",
        "vendor_name": "Wireshark",
        "event_type": event_type, # Determinato dinamicamente
        "description": f"Packet capture. Protocols: {frame.get('frame.protocols', 'N/A')}..."
    }
}
# Le sezioni principal, target, network, about, application_protocol_data, additional
# vengono popolate e aggiunte a udm_payload solo se contengono dati effettivi.
# Esempio per la sezione 'principal':
cleaned_principal = clean_none_values(udm_principal) # clean_none_values rimuove chiavi con valore None
if cleaned_principal: udm_payload["principal"] = cleaned_principal
# ... logica simile per le altre sezioni ...
return {"event": udm_payload} # L'evento UDM finale e' incapsulato in un oggetto "event"

```

Listing 3: Estratto da `json2udm_cloud.py`: Struttura di un evento UDM.

- **Estrazione Dati Migliorata:** Utilizzo di funzioni helper come `get_nested_value` per un'estrazione sicura dei valori da strutture JSON potenzialmente complesse (evitando `KeyError`) e `extract_values_from_tshark_section` per campi multi-valore (es. query DNS).
- **Logging Dettagliato:** Lo script produce log informativi, inclusi contatori di pacchetti processati ed errori (`UDM_PACKETS_PROCESSED`, `UDM_PACKET_ERRORS`), utili per il monitoraggio tramite le metriche basate su log (LBM).

Il processo di conversione per ogni pacchetto implica l'estrazione dei layer protocollari (frame, eth, ip, tcp, udp, dns, http, tls, etc.), la mappatura dei campi rilevanti nelle sezioni UDM appropriate, e la normalizzazione dei dati. La selezione dei campi mappati rimane focalizzata sulla rilevazione di minacce e anomalie di rete.

4 Componente Sniffer (Edge)

Lo Sniffer è l'agente di cattura della pipeline, progettato per operare in ambienti on-premises o su dispositivi edge. È implementato come un container Docker leggero basato su Alpine Linux, contenente `tshark` e gli strumenti `gcloud`.

Il suo comportamento è orchestrato dallo script `sniffer_entrypoint.sh`. All'avvio, lo script valida le variabili d'ambiente (ID progetto, bucket, topic, e un `SNIFFER_ID` univoco), si autentica con GCP, e rileva automaticamente un'interfaccia di rete attiva (escludendo interfacce virtuali o di loopback). Avvia quindi `tshark` in background, configurato per la rotazione dei file di cattura (supportando estensioni `.pcap` e `.pcapng` tramite il pattern `*.pcap*`) in base a dimensione o tempo (configurabile tramite la variabile `ROTATE`).

```
# La variabile $INTERFACE viene determinata automaticamente o presa dall'ambiente.
# $ROTATE definisce i criteri di rotazione (es., "-b filesize:10240 -b duration:60").
# $LIMITS puo' contenere filtri di cattura aggiuntivi.
# I file vengono scritti nella directory $CAPTURE_DIR con un nome base $FILENAME_BASE.
echo "(ID: $SNIFFER_ID) Starting tshark capture..."
tshark $INTERFACE $ROTATE $LIMITS -w "$CAPTURE_DIR/$FILENAME_BASE.pcap" &
TSHARK_PID=$! # Salva il PID di tshark per monitoraggio e shutdown.
echo "(ID: $SNIFFER_ID) tshark started with PID $TSHARK_PID"
```

Listing 4: Estratto da `sniffer_entrypoint.sh`: Avvio di `tshark`.

Un loop di monitoraggio verifica costantemente la presenza di file completati (non più attivamente scritti da `tshark`, identificati tramite `lsof`). Per ogni file completato, lo script ne logga la dimensione (`PCAP_SIZE_BYTES`), lo carica su GCS, pubblica una notifica con il nome del file su Pub/Sub, e infine rimuove il file locale. L'output dei log dello sniffer, come mostrato nello screenshot (Figura 4), evidenzia queste operazioni.

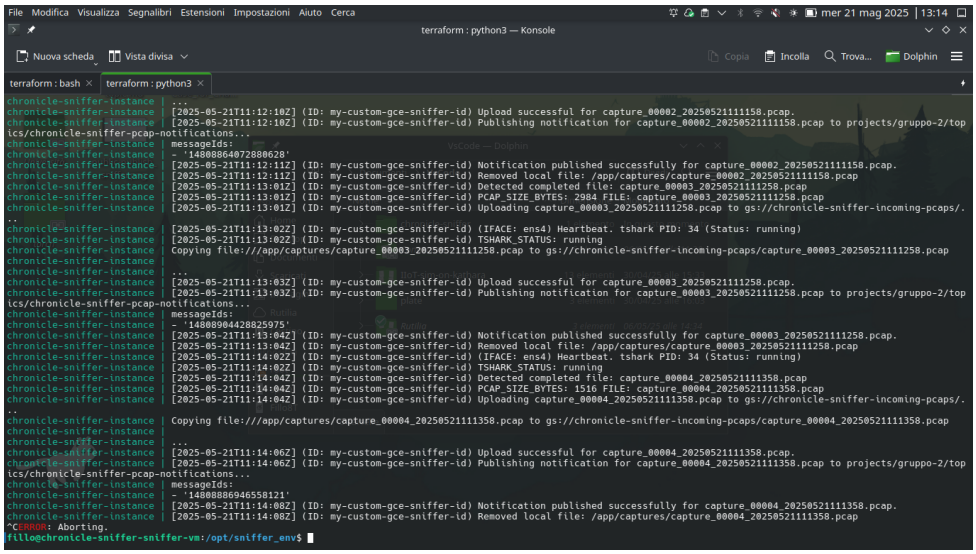

Figura 4: Log del container Sniffer durante l'upload e la notifica.

```
# Questo blocco viene eseguito per ogni file PCAP completato.
# "$pcap_file_path" e' il percorso del file locale.
# "$current_pcap_file_basename" e' solo il nome del file.
```

```
echo "[$(date)] (ID: $SNIFFER_ID) Uploading $current_pcap_file_basename to gs://${INCOMING_BUCKET}/..."
if gcloud storage cp "$pcap_file_path" "gs://${INCOMING_BUCKET}/" --project "$GCP_PROJECT_ID"; then
    echo "[$(date)] (ID: $SNIFFER_ID) Upload successful for $current_pcap_file_basename."

    echo "[$(date)] (ID: $SNIFFER_ID) Publishing notification for $current_pcap_file_basename to ${PUBSUB_TOPIC_ID}..."
    if gcloud pubsub topics publish "$PUBSUB_TOPIC_ID" --message "$current_pcap_file_basename" --project "$GCP_PROJECT_ID"; then
        echo "[$(date)] (ID: $SNIFFER_ID) Notification published successfully for $current_pcap_file_basename."
        processed_files+=("$current_pcap_file_basename") # Aggiunge alla lista dei file processati.
        rm "$pcap_file_path" # Rimuove il file locale.
        echo "[$(date)] (ID: $SNIFFER_ID) Removed local file: $pcap_file_path"
    else
        # Errore nella pubblicazione Pub/Sub, il file non viene rimosso e verra' ritentato.
        echo "[$(date)] (ID: $SNIFFER_ID) Error: Failed to publish notification for $current_pcap_file_basename. Will retry."
    fi
else
    # Errore nell'upload GCS, il file non viene rimosso e verra' ritentato.
    echo "[$(date)] (ID: $SNIFFER_ID) Error: Failed to upload $current_pcap_file_basename to GCS. Will retry."
fi
```

Listing 5: Estratto da `sniffer_entrypoint.sh`: Upload a GCS e notifica a Pub/Sub.

Lo script include una funzione di *heartbeat* che logga periodicamente lo stato di `tshark` (`TSHARK_STATUS: running/stopped`) e l'ID dello sniffer, facilitando il monitoraggio remoto della sua attività. Gestisce anche i segnali *SIGTERM* e *SIGINT* per un arresto pulito di `tshark`. La configurazione per l'avvio locale tramite `docker-compose` è fornita per facilitare test e sviluppo.

Un aspetto cruciale dello sniffer è il suo basso impatto sulle risorse del sistema edge. Come evidenziato dall'output del comando `docker stats` (vedi Figura 5), il container dello sniffer opera con un utilizzo minimo di CPU e memoria (nell'esempio, circa 0.05% CPU e ∼100MB di RAM), rendendolo facilmente integrabile anche in ambienti con risorse limitate senza impattare significativamente le prestazioni di altri servizi. Questa efficienza è fondamentale per un'adozione su larga scala in contesti aziendali.

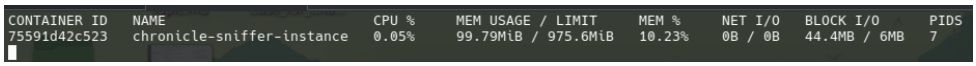

Figura 5: Output di `docker stats` per il container Sniffer.

5 Componente Processor (Cloud Run)

Il Processor, eseguito su Cloud Run, è il cuore dell'elaborazione. L'applicazione Flask `processor_app.py` riceve le notifiche Pub/Sub. Una novità importante è la verifica "*lazy*" dell'accessibilità ai bucket GCS all'avvio dell'istanza o alla prima richiesta, per gestire meglio eventuali ritardi nella propagazione dei permessi IAM post-deployment.

Per ogni notifica, l'applicazione scarica il file di cattura indicato, lo elabora in una directory temporanea, e orchestra le due fasi di conversione:

1. PCAP → JSON: Esegue `tshark -T json` come subprocess.
2. JSON → UDM: Esegue lo script `json2udm_cloud.py` (anch'esso come subprocess) sull'output JSON di `tshark`.

I log del servizio Cloud Run, visibili nello screenshot (Figura 6), mostrano le fasi di conversione e l'eventuale upload del file UDM risultante.

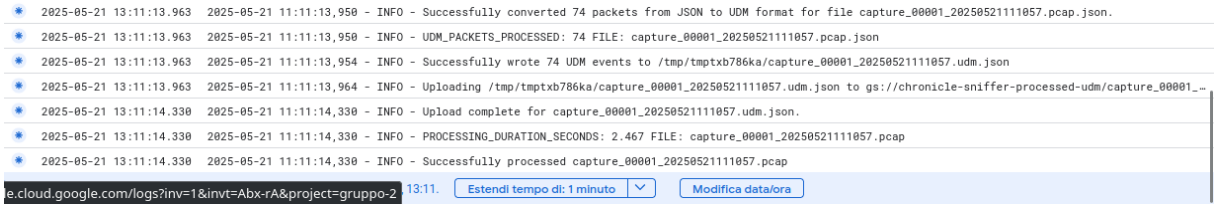

Figura 6: Log del servizio Cloud Run (Processor) durante l'elaborazione.

```python
# Questa logica viene eseguita all'interno di un blocco try...except per la gestione degli errori.
# local_pcap_path, local_json_path, local_udm_path sono percorsi in una directory temporanea.
try:
    # ... (download del file pcap da GCS) ...

    logging.info(f"Converting {local_pcap_path} to JSON...")
    tshark_command = ["tshark", "-r", local_pcap_path, "-T", "json"]
    # L'output JSON di tshark viene scritto direttamente su file.
    with open(local_json_path, "w") as json_file:
        # subprocess.run esegue il comando. check=True fa si che venga sollevata
        # un'eccezione CalledProcessError se tshark restituisce un exit code non zero.
        process = subprocess.run(tshark_command, stdout=json_file, stderr=subprocess.PIPE, text=True, check=True)
    logging.info(f"tshark conversion successful: {local_json_path}")
    if process.stderr: logging.warning(f"tshark stderr: {process.stderr.strip()}")

    logging.info(f"Converting {local_json_path} to UDM: {local_udm_path}")
    udm_script_command = ["python3", "/app/json2udm_cloud.py", local_json_path, local_udm_path]
    # Anche qui, check=True per catturare errori dallo script di conversione UDM.
    # capture_output=True per raccogliere stdout e stderr dello script.
    process = subprocess.run(udm_script_command, capture_output=True, text=True, check=True)
    logging.info(f"UDM conversion script done for {pcap_filename}.")
    if process.stdout: logging.info(f"json2udm_cloud.py stdout: {process.stdout.strip()}")
    if process.stderr: logging.warning(f"json2udm_cloud.py stderr: {process.stderr.strip()}")

    # ... (upload del file UDM a GCS) ...

except subprocess.CalledProcessError as e:
    # Gestisce errori specifici dei subprocess (tshark o json2udm_cloud.py).
    error_message = e.stderr.strip() if e.stderr else e.stdout.strip()
    logging.error(f"Subprocess error: CMD: {' '.join(e.cmd)} ERR: {error_message}", exc_info=False)
    return "Internal Server Error during processing step.", 500 # Segnala a Pub/Sub di ritentare.
# ... (altri blocchi except per errori GCS, etc.)
```

Listing 6: Estratto da `processor_app.py`: Chiamata ai subprocess per `tshark` e `json2udm_cloud.py`.

L'applicazione logga la durata dell'elaborazione (`PROCESSING_DURATION_SECONDS`) per ogni file e gestisce gli errori restituendo codici HTTP appropriati a Pub/Sub per controllare i tentativi di acknowledgement (ack) o negative acknowledgement (nack).

6 Infrastruttura come Codice (IaC) con Terraform

L'intera infrastruttura GCP è gestita tramite Terraform. La configurazione è modulare e include:

- Service Account dedicati con permessi minimi.
- Bucket GCS per i file di input e output.
- Topic Pub/Sub principale e un Dead-Letter Topic (DLQ), con una sottoscrizione *push* verso Cloud Run configurata con autenticazione OIDC e una politica di *dead-lettering*.

```
resource "google_pubsub_subscription" "processor_subscription" {
  project              = var.gcp_project_id
  name                 = "${var.base_name}-processor-sub"
  topic                = module.pubsub_topic.topic_id # Topic principale
  ack_deadline_seconds = 600 # Tempo per Cloud Run per processare

  push_config {
    push_endpoint = module.cloudrun_processor.service_url # URL del servizio Cloud
    Run
    # Configurazione OIDC per l'autenticazione sicura
    dynamic "oidc_token" {
      for_each = !var.allow_unauthenticated_invocations ? [1] : []
      content {
        service_account_email = google_service_account.cloud_run_sa.email
        audience              = module.cloudrun_processor.service_url
      }
    }
  }

  # Politica per inviare messaggi non elaborabili al DLQ
  dead_letter_policy {
    dead_letter_topic     = module.pubsub_topic.dlq_topic_id # Topic DLQ
    max_delivery_attempts = 5 # Numero massimo di tentativi prima del DLQ
  }
  # ... (depends_on per gestione dipendenze)
}

```

Listing 7: Estratto da `terraform/main.tf`: Definizione della sottoscrizione Pub/Sub.

- Servizio Cloud Run per il processor, con configurazione di immagine, variabili d'ambiente, risorse (CPU, memoria impostata a `2Gi` per gestire meglio `tshark`), concorrenza e *health probes*.

```
resource "google_cloud_run_v2_service" "processor" {
  project  = var.project_id
  name     = var.service_name
  location = var.region

  template {
    service_account = var.service_account_email
    max_instance_request_concurrency = var.max_concurrency
    timeout                          = "600s" # Timeout per richiesta
    containers {
      image = var.image_uri
      ports { container_port = 8080 } # Porta interna del container
      dynamic "env" { # Passaggio dinamico delle variabili d'ambiente
        for_each = var.env_vars
        content {
          name  = env.key
          value = env.value
        }
      }
      resources { # Limiti di CPU e memoria per istanza
        limits = {
          cpu    = var.cpu_limit
          memory = var.memory_limit # Es. "2Gi"
```

```
        }
      }
      # Probe per monitorare la salute dell'applicazione
      startup_probe { http_get { path = "/" } # ... (configurazione dettagliata omessa per brevità) }
      liveness_probe { http_get { path = "/" } # ... (configurazione dettagliata omessa per brevità) }
    }
  }
  # ... (configurazione traffico omessa)
}

```

Listing 8: Estratto da `terraform/modules/cloudrun_processor/main.tf`

- **Metriche Basate su Log (LBMs):** Definite in `terraform/main.tf` per tracciare eventi chiave come *heartbeat* dello sniffer, upload di PCAP, errori di pubblicazione Pub/Sub, successo/fallimento delle conversioni `tshark`, pacchetti UDM processati/errori, e latenza di elaborazione. Queste metriche alimentano la dashboard operativa.
- **Dashboard Operativa:** Un file JSON (`terraform/dashboards/main_operational_dashboard.json`) definisce una dashboard completa in Cloud Monitoring utilizzando Monitoring Query Language (MQL). Questa dashboard visualizza le LBMs e le metriche standard dei servizi GCP, offrendo una visione centralizzata dello stato della pipeline. (Figure 7).

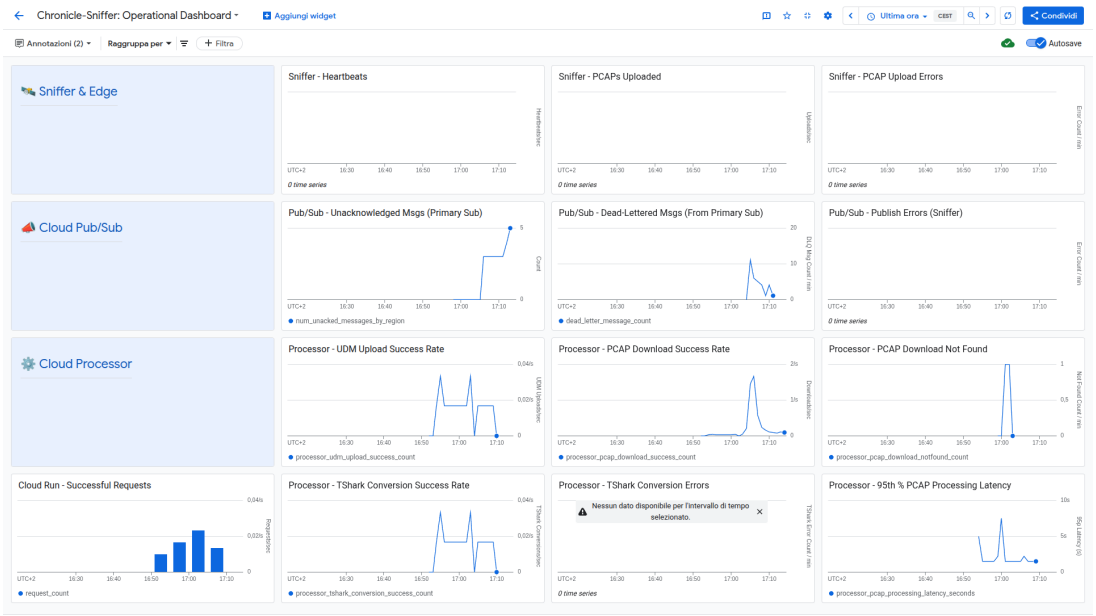

Figura 7: Dashboard Operativa in Cloud Monitoring.

- **Politica di Alert (Esempio):** Una politica di alert di esempio per gli sniffer inattivi (basata sulla metrica di *heartbeat*) dimostra come estendere l'osservabilità.
- **VM di Test (Opzionale):** Un modulo per creare una VM GCE che simula un ambiente on-prem, con uno script di avvio che prepara l'ambiente Docker Compose per lo sniffer.

7 Valore Didattico ed Evoluzione del Progetto

Questo progetto si distingue per la sua evoluzione da una soluzione locale a un'architettura cloud-nativa sofisticata. Il passaggio chiave è stato l'offloading dell'elaborazione intensiva al

cloud e l'adozione di un design *event-driven* e serverless. L'introduzione del parsing JSON in streaming (`ijson`) nello script `json2udm_cloud.py` è un esempio di ottimizzazione per sfruttare a pieno l'ambiente cloud, dovuta ai limiti di memoria di Cloud Run. La definizione dell'intera infrastruttura e della dashboard di monitoraggio come codice (Terraform) sottolinea le moderne pratiche DevOps per la gestione di sistemi scalabili e affidabili. Il progetto dimostra l'applicazione pratica di concetti come disaccoppiamento, resilienza (DLQ), sicurezza IAM e osservabilità end-to-end. Le competenze acquisite includono la gestione di infrastrutture complesse con IaC, il design di sistemi serverless, il debugging in ambienti distribuiti e l'ottimizzazione delle risorse cloud.

8 Aspetti Operativi

L'architettura progettata integra diversi meccanismi per garantire scalabilità, affidabilità, sicurezza e osservabilità, aspetti cruciali per un servizio operativo.

La **scalabilità** è gestita principalmente da Cloud Run, che adatta automaticamente il numero di istanze del Processor al carico di messaggi Pub/Sub, e dai servizi GCP sottostanti che scalano in modo trasparente. L'**affidabilità** si basa sul disaccoppiamento fornito da Pub/Sub, che garantisce la persistenza dei messaggi e gestisce i tentativi di consegna in caso di fallimenti temporanei del Processor. Cloud Run contribuisce ulteriormente monitorando la salute delle istanze tramite *health probes* e sostituendo quelle problematiche. La **sicurezza** è implementata attraverso il principio del *least privilege* per i Service Account, sull'autenticazione OIDC tra Pub/Sub e Cloud Run, la gestione delle chiavi SA on-premises e la configurazione sicura dei bucket GCS. Infine, l'**osservabilità** è un punto di forza di questa versione evoluta. Oltre ai log centralizzati in Cloud Logging, il sistema si avvale di strumenti di monitoraggio avanzati:

- **Metriche Basate su Log (LBMs):** Queste metriche, definite via Terraform, catturano eventi specifici dell'applicazione (es. `sniffer_heartbeat_count`, `pcap_files_uploaded_count`, `processor_pcap_processing_latency_seconds`, `processor_udm_packets_processed_count`). Sono essenziali per comprendere il comportamento interno e le prestazioni della pipeline.
- **Dashboard Operativa Personalizzata:** Anch'essa gestita come codice (`json`) e deployata da Terraform, utilizza MQL per visualizzare sia le LBMs sia le metriche standard dei servizi GCP. Offre una visione d'insieme dello stato della pipeline, delle prestazioni dei singoli componenti (sniffer, Pub/Sub, processor) e dei tassi di errore, cruciale per la manutenzione proattiva e il troubleshooting.

9 Test e Validazione

La validazione della pipeline è facilitata dalla Test Generator VM, un componente cruciale dell'infrastruttura definita tramite Terraform. Questa VM non è semplicemente un ambiente di test generico, ma è concepita specificamente per simulare un ambiente on-premises o edge in cui lo sniffer opererebbe tipicamente. Questo ambiente simulato è fondamentale perché permette di replicare le condizioni operative reali del componente Sniffer, che è l'unico elemento della pipeline destinato a operare al di fuori dell'infrastruttura GCP gestita. Lo script di avvio della VM (`startup_script_vm.sh`) automatizza la configurazione di un ambiente Docker Compose completo, pronto per eseguire il container dello sniffer. In questo ambiente simulato, è possibile utilizzare strumenti come `tcpreplay` per "riprodurre" file PCAP preesistenti (ad esempio, un file illustrativo come `synflood_capture.pcap` contenente traffico di un attacco simulato), immettendo traffico controllato nella rete della VM che verrà catturato dallo sniffer.

```
# Questo script, eseguito all'avvio della VM di test, automatizza la configurazione
# dell'ambiente per lo sniffer. Recupera i parametri necessari dai metadati dell'istanza
```

```
# (passati da Terraform) e crea i file di configurazione per Docker Compose.

# ... (installazione docker, pull immagine dello sniffer) ...

echo "Creating .env file for the sniffer in $SNIFFER_ENV_FILE"
# Il file .env conterra' le variabili d'ambiente specifiche per questa istanza
# dello sniffer sulla VM di test.
cat << EOF_ENV > "$SNIFFER_ENV_FILE"
GCP_PROJECT_ID=$VM_GCP_PROJECT_ID_FROM_METADATA
INCOMING_BUCKET=$VM_INCOMING_BUCKET_FROM_METADATA
PUBSUB_TOPIC_ID=$VM_PUBSUB_TOPIC_ID_FROM_METADATA
SNIFFER_ID=\${VM_SNIFFER_ID_FROM_METADATA}
# GCP_KEY_FILE e' ${SNIFFER_GCP_KEY_CONTAINER_PATH}/key.json
GCP_KEY_FILE=\${SNIFFER_GCP_KEY_CONTAINER_PATH}/key.json # Path interno al container
EOF_ENV

# ... (creazione di docker-compose.yml e docker-compose.override.yml specifici per la VM)
    ...
# L'override.yml gestisce il montaggio della chiave SA e della directory delle catture.
```

Listing 9: Estratto da `terraform/modules/test_generator_vm/startup_script_vm.sh`.

In aggiunta, per testare la reazione del sistema a scenari di traffico specifici o attacchi simulati, è possibile installare e utilizzare sulla VM strumenti come `hping3`. Ad esempio, un attacco SYN flood può essere simulato per osservare come i pacchetti anomali vengono catturati, processati e rappresentati nel formato UDM.

```
# Questo comando invia un gran numero di pacchetti TCP SYN alla porta 80
# dell'IP target, simulando l'inizio di molte connessioni senza completarle.
sudo hping3 -S --flood -p 80 <IP_TARGET_NELLA_RETE_DELLA_VM>
```

Listing 10: Esempio di comando `hping3` per simulare un SYN flood (da eseguire sulla VM di test)

Questa capacità di simulare un ambiente on-premises realistico, completo di strumenti per la generazione e la riproduzione di traffico, è fondamentale per validare end-to-end il funzionamento della pipeline e per osservarne il comportamento in condizioni controllate, inclusa la risposta a potenziali minacce di rete. File di esempio come un ipotetico `synflood_capture.pcap` e il corrispondente `synflood_capture.udm.json` servirebbero come riferimento per questa validazione.

10 Conclusioni e Sviluppi Futuri

Questo progetto ha raggiunto l'obiettivo di evolvere una semplice utility di conversione in una pipeline di analisi del traffico di rete robusta, scalabile, osservabile e gestita come codice su GCP. L'adozione di tecniche come il parsing in streaming, l'architettura *event-driven* con DLQ, e l'osservabilità tramite dashboard IaC dimostrano una comprensione matura dei principi per la creazione di servizi cloud affidabili.

Possibili sviluppi futuri includono:

- Implementazione di test automatici di integrazione e unitari.
- Integrazione diretta con l'API di ingestion di *Chronicle Security Operations* (*Google SecOps*).

Il sistema attuale, tuttavia, costituisce una solida piattaforma per l'analisi del traffico di rete su scala.

11 Bibliografia e Sitografia

Riferimenti bibliografici

[1] Lucchesi, Filippo (2025). *chronicle-Sniffer*. GitHub Repository. https://github.com/fillol/chronicle-sniffer (Codice Sorgente)

[2] Lucchesi, Filippo (2024). *Wireshark-to-Chronicle-Pipeline*. GitHub Repository. https://github.com/fillol/Wireshark-to-Chronicle-Pipeline (Progetto originario di Cybersecurity)

[3] Immagine Sniffer su Dockerhub. https://hub.docker.com/r/fillol/chronicle-sniffer

[4] Google Cloud. Cloud Run. https://cloud.google.com/run/docs

[5] Google Cloud. Pub/Sub. https://cloud.google.com/pubsub/docs

[6] Google Cloud. Cloud Storage. https://cloud.google.com/storage/docs

[7] Google Cloud. Unified Data Model (UDM). https://cloud.google.com/chronicle/docs/unified-data-model/udm-overview

[8] Google Cloud. Cloud Monitoring. https://cloud.google.com/monitoring/docs

[9] Google Cloud. Identity and Access Management (IAM). https://cloud.google.com/iam/docs

[10] Google SecOps. Chronicle https://cloud.google.com/security/products/security-operations

[11] Terraform by HashiCorp. https://developer.hashicorp.com/terraform/docs

[12] Docker. https://docs.docker.com/

[13] TShark Manual Page. https://www.wireshark.org/docs/man-pages/tshark.html

[14] Python Software Foundation. ijson https://pypi.org/project/ijson/

[15] Hping3. https://www.kali.org/tools/hping3/

[16] Tcpreplay Manual Page. https://tcpreplay.appneta.com/wiki/tcpreplay-man.html