Mapping Amazon Application Load Balancer Access Logs to the Open Cybersecurity Schema Framework (OCSF)

Contributed by: Jonathan Rau & Aurora Starita

Table of Contents

Introduction to OCSF

In this whitepaper, we'll focus on mapping **Amazon Application Load Balancer (ALB)** access logs to the **Open Cybersecurity Schema Framework (OCSF)** format.

OCSF is a standardized schema designed to help organizations normalize and correlate data from different sources, making it easier to analyze and act on cybersecurity events. For more information on using OCSF see our **Definitive Guide to OCSF Mapping** and our **Absolute Beginner's Guide to OCSF**.

Mapping ALB access logs to OCSF—along with other OCSF-normalized log sources—will help to simplify investigations, incident response (IR), threat hunting, security analytics, and other security- and observability-relevant use cases.

This holds true due to the fact that OCSF is a strongly-typed, hierarchical, and actively maintained schema that provides normalized and standardized locations to map and transform all security data—Amazon ALB access logs in this case.

OCSF is also well optimized for storage in columnar data formats such as Apache Parquet for ingestion into your data warehouses, data lakes, open lakehouses, and even your Security Information & Event Management (SIEM) solutions such as Splunk, or SIEMs built atop Amazon OpenSearch Service.

Making a mapped schema is an attractive option not just for cost-conscious organizations, but for those who want to take control of their data. Which we think you should.

What Is an AWS Application Load Balancer?

AWS ALB is a managed service by AWS that distributes incoming application traffic across multiple targets, such as EC2 instances, containers, or Lambda functions.

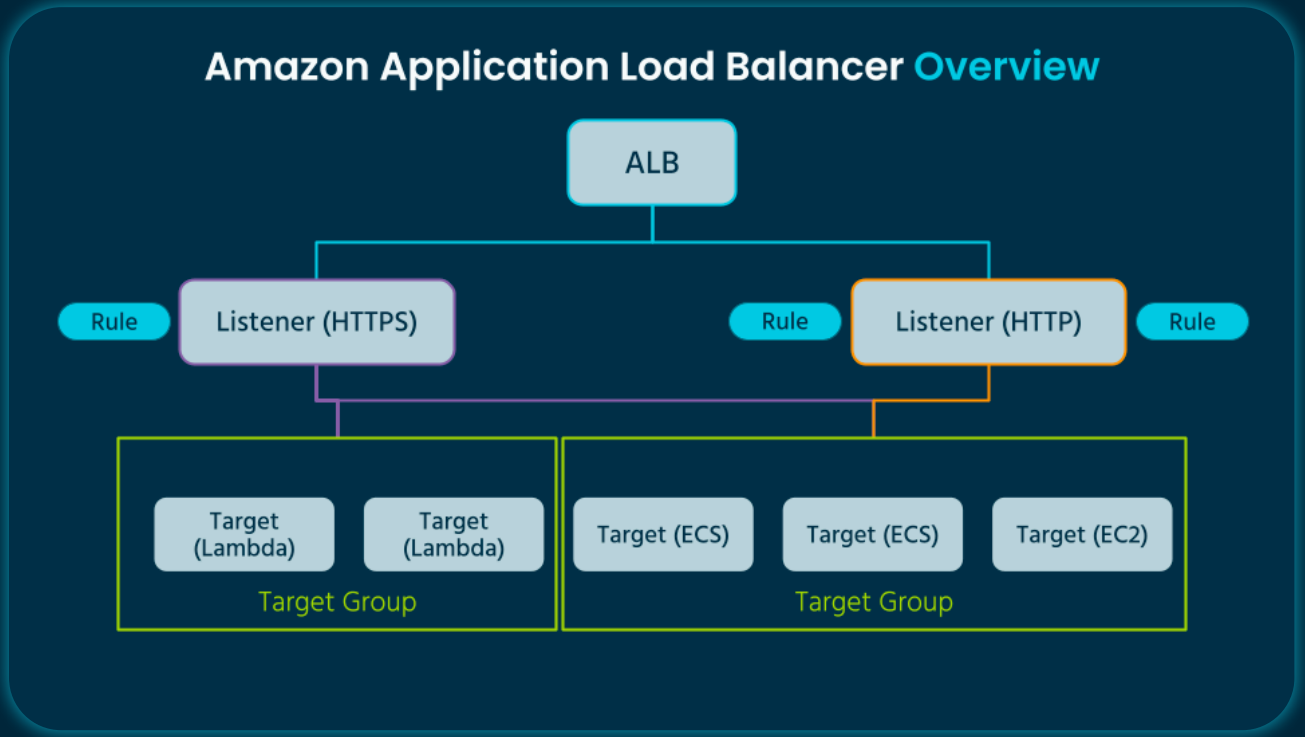
ALB is part of the larger Amazon Elastic Load Balancing (ELBv2) service that includes ALB for OSI Layer 7 (HTTP/HTTPS/Websockets) traffic and Network Load Balancer (NLB) for OSI Layer 4 (TCP/UDP/TLS) traffic.

Each ALB has one or more **listeners** which map to a protocol and port (e.g., HTTP on port 8080, or HTTPS on port 443) which use **rules** to route requests to registered **target groups** which is a collection of one or more targets.

Targets include Amazon EC2 instances and AWS Lambda functions, among others. For more information on ALB refer to the **Application Load Balancer section** of the Amazon Elastic Load Balancing documentation.

ALB also provides the ability to apply Distributed Denial of Service (DDOS) protection to targets behind it via AWS Shield Advanced, even more effectively when used in conjunction with AWS Web Application Firewall (AWS WAFv2).

There are controls to drop invalid HTTP headers as well as protect against HTTP desync attacks, and the rules engine allows for finer grained routing and health checks.



Most important for our (security-relevant) purposes: it provides detailed access logs that capture information about requests and their outcomes. These access logs are a good source of data for performance monitoring and security analysis.

Using a service such as Amazon Security Lake can help centralize AWS WAFv2 access logs and Amazon Virtual Private Cloud (VPC) flow logs, where this solution will help enable you to operationalize around the ALB access logs as well.

ALB access logs provide:

- The context on the source IP and port
- The specific resource your ALB routed them to
- Outcomes such as if the request was allowed
- Payload times and sizes

Plus ALB-specific data such as:

- Which Server Name Indicator (SNI) and Amazon Certificate Manager (ACM) X.509 certificates were used
- HTTP Desync protection classification actions and reasons

On its own, the data may not be too exciting. However, when used in conjunction with WAF, VPC Flow Logs, and (optionally) AWS Global Accelerator and/or Amazon CloudFront access logs,it provides the “full story” of connectivity–and potential malicious activity–in your AWS network stack.

Challenges in Mapping ALB Logs to OCSF

Mapping ALB access logs to OCSF isn't without its difficulties:

- **Log Format:** ALB access logs have a space-separated log format that also uses quotations around certain elements. This makes parsing via simple delimiting impossible, and even makes using regular expressions (regex) complicated.
- **Bucket Policies by Region:** Log files are stored in Amazon S3 buckets with each legacy AWS Regional log delivery service (any region created before August 2021) requiring a specific Bucket Policy and for the Bucket to be created in that specific region. If you are multi-regional, you require one bucket per Account wherever ALBs are deployed, each with its own Bucket Policy allowing the specific AWS account of the managed Log Delivery service.
- **Centralization Difficulties:** While this paper focuses on mapping and normalizing logs, we are *not addressing log centralization*. Centralizing logs across multiple regions and accounts requires additional tooling and policies that go beyond the scope of this guide.

So it's a challenge but so what? Are you afraid of a little challenge? We're not.

This paper aims to provide a practical guide for organizations looking to extract more value from their ALB logs by mapping them to OCSF and to get started, we'll stick with single buckets.

In this paper, you will learn the anatomy of an ALB log and common processing pitfalls within them. You will learn the overall mechanism for Extraction, Transformation, and Loading (ETL) of ALB access logs into OCSF format, and do it yourself with Python. Finally, you will learn how to analyze basic data within the ALB access logs using DuckDB.

Prerequisites

For the best experience, download VSCode and ensure you have at least Python 3.11 installed on your machine along with your preferred package manager such as Pip, Poetry or otherwise. VSCode handles setting up virtual environments as well as running scripts as notebooks.

When using the script for this paper, VSCode will prompt you to install jupyter and other dependencies. You can run any script as a notebook by adding the following characters above your various code "blocks": # %%.

To install DuckDB, refer to the official DuckDB Installation section of their **documentation**. The simplest way to install it is on a MacOS using homebrew with: brew install duckdb.

After DuckDB is installed, you will need to install several Python libraries using your preferred package manager such as pip or otherwise. Our code repo contains a requirements.txt file here that lists them all.

If you need it, take our crash course on **SQL and DuckDB and its applicability to Security Operations teams**.

Finally, you can download the script to process ALB logs and the notebook-ified DuckDB script from our **GitHub repository**. The rest of the paper will assume you have the code and the ability to run it, including AWS IAM credentials that allow you to download objects from an S3 bucket in your region.

Anatomy of an ALB Access Log

ALB access logs are space-delimited logs—similar to Apache Common Event Format (CEF) and AWS VPC flow logs—with specialized log fields escaped with double quotes. For a full description of every possible log field, see the **Syntax subsection of the Access logs for your Application Load Balancer section** of the Elastic Load Balancing documentation.

Double quotes are used to encapsulate fields that naturally have white space in them such as the request field which contains the HTTP Method (e.g., GET or POST), the URL of the target, and the HTTP Version (e.g., HTTP/1.0 or HTTP/2.0). For example, consider this request field from one of our honeypots: `"GET http://mock-lambda-external-alb-redacted.us-east-2.elb.amazonaws.com:80/ HTTP/1.0"`

Likewise the user-agent field is also encapsulated in double quotes `"vodafone/1.0/V802SE/SEJ001 Browser/SEMC-Browser/4.1"` as are other fields. As mentioned before, this makes parsing the ALB logs with easy mechanisms such as splitting them by white space difficult.

```
http 2024-11-16T23:59:49.445081Z app/mock-lambda-external-alb/
redacted 5.8.11.202:60000 - 0.012 0.034 0.000 502 - 96 272 "GET http://
mock-lambda-external-alb-redacted.us-east-2.elb.amazonaws.com:80/
HTTP/1.0" "vodafone/1.0/V802SE/SEJ001 Browser/SEMC-Browser/4.1" - -
arn:aws:elasticloadbalancing:us-east-2:redacted123:targetgroup/mock-lambda-
tg/123redactedlol123 "Root=1-673931f5-3c6b885a0d99945f6cdc880d" "-" "-" 0
2024-11-16T23:59:49.399000Z "waf,forward" "-" "LambdaInvalidResponse" "-"
"-" "-" "-" TID_5c70dac1416ce84c9b5eee0d785cf295n
```

An example of a real world HTTP log from one of our honeypots will look like this. (HTTPS logs are inherently more detailed.)

Writing a broadly applicable regular expression is also difficult due to the fact that there can be missing information within the logs, or places where there are comma delimited fields within double quoted fields.

For instance, anytime there is a missing entry for a log field, it is replaced with a single dash (–). Likewise, there are fields such as actions_executed which can contain multiple entries such as `"waf,forward"` which denote that the specific connection was inspected by a WAF before being sent to the target group with the forward action. These actions can differ based on your rules.

There are some regex patterns online that will work in 90% of cases, and if the final destination was a SIEM or another log management system where rich text search could be used against indexed data this parsing inconsistency may not be an issue.

However, for storing this data as structured information with your open security data lakehouse or within another data lake, a higher level of due care is required to avoid data quality issues.

For our purposes Grok–which is built atop regex at a higher level of abstraction–is a better tool as, though it may be less flexible than regex overall, it's easier to work with and optimized for the exact kind of pattern matching we're looking to do. You will see how the Grok pattern is written in later sections of this paper.

Furthermore, there are concatenated fields within the access log that contain important data. For instance, both the client:port and target:port fields combine the IP address and the port with a colon (:) delimiter. There are also cases where target:port is missing due to the type of downstream target, such as an AWS Lambda function.

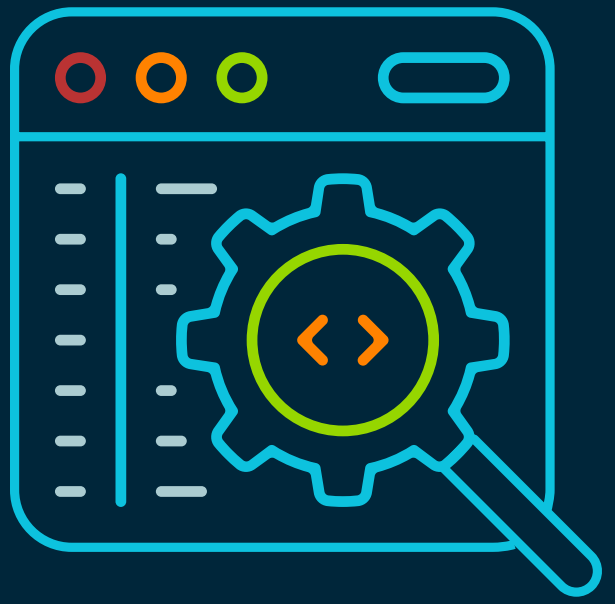
There are also cases where further processing is required to pull out other pertinent information that is not otherwise purposely concatenated such as the client:port field. In this case, the target_group_arn field contains the Amazon Resource Name (ARN) of the target group the traffic was routed to, or was intended to be routed to.

The ARN also contains the AWS account ID and the AWS region name which is not otherwise present in the log. This value can be processed to extract those values, it is incredibly important for attribution and ownership, especially in very large multi-account and multi-region AWS environments.

There are duplicate fields within the ALB access log, such as target:port_list, which was intended to hold the IP:port pairs for target groups with several targets with mixed IP addresses, such as Amazon EC2 instances. However, this field is a duplicate of the target:port field, and is unnecessary to retain.

Likewise, there are very rarely used fields such as classification and classification_reason, which contain details about HTTP desync attacks according to the RFC 7230 standard. If HTTP desync attack prevention is important to your application security efforts, or to your threat model(s), you should consider retaining these but otherwise you can consider dropping their inclusion altogether.

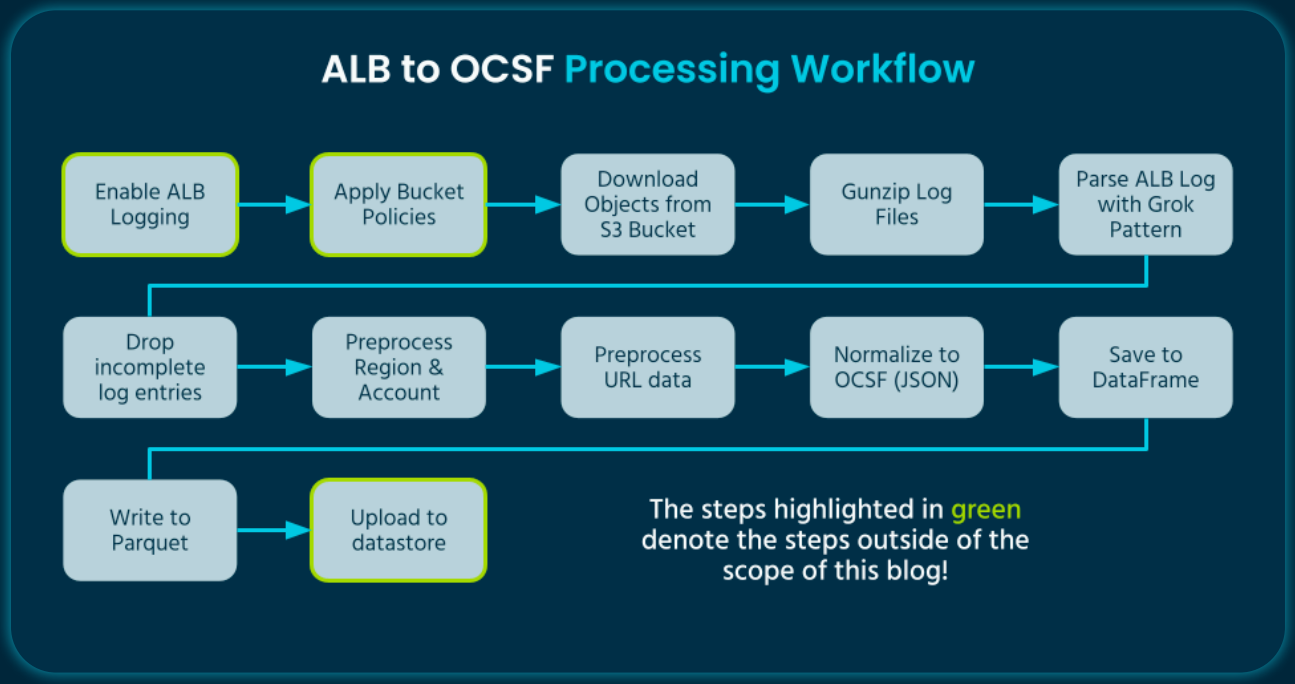
In the next section, you will learn about how to effectively process these access logs and transform them into an OCSF-formatted Parquet file to ultimately use for analysis with DuckDB or ingest into your open security data lakehouse.



Processing ALB Access Logs Overview

Processing and packaging (not in the sausage or SBOM way), the ALB Access Logs into usable OCSF data is a multi-step process. This is also known as Extraction, Transformation, and Loading (ETL) data, and is a foundational SecDataOps skillset to know. The most important prerequisite being that you have ALBs with live traffic that are publishing logs into an S3 bucket, and the bucket can receive the logs.

Referring to the below diagram, the high-level steps for processing raw ALB access logs into proper OCSF formatted logs are as follows.



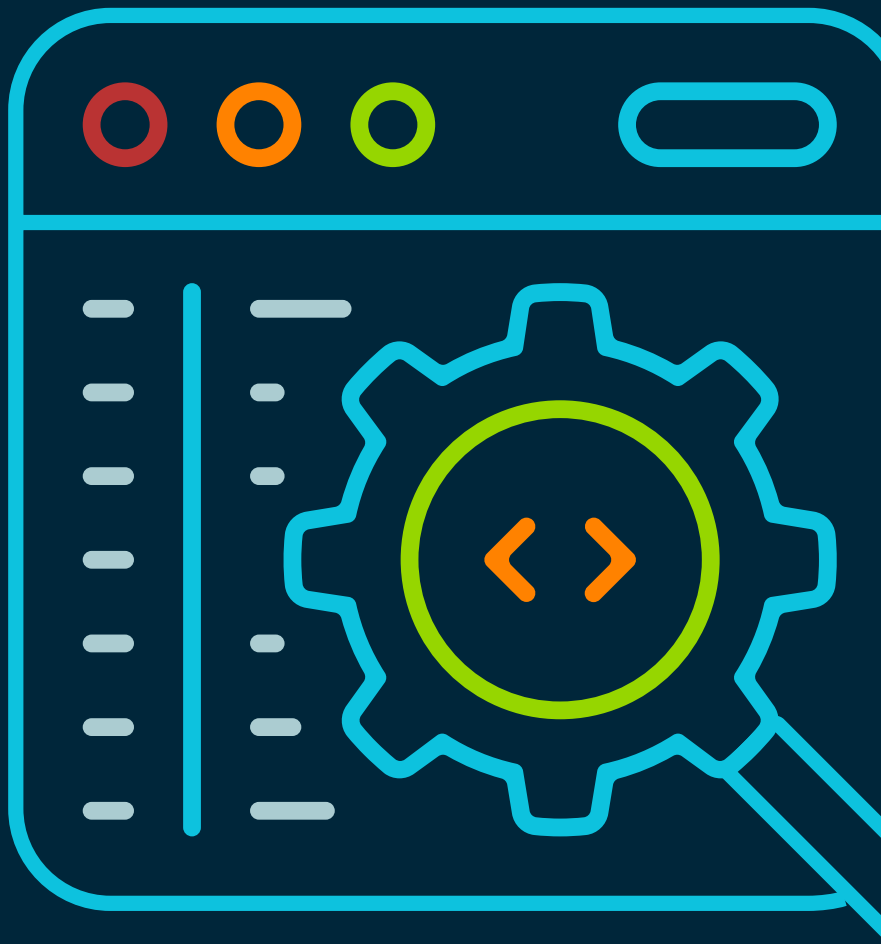
1. The AWS-managed Log Delivery service for ALB delivers access logs to designated S3 bucket(s) they are given permission to in a variable time frame.
2. The script downloads the raw log files—which are GZIP'ed text files—from a specific bucket(s) or certain prefixes within the bucket(s).
3. Log files have their GZIP compression removed ("gunzipped") and the raw log is parsed with a Grok pattern to convert the text based log into a Python dictionary.
4. Incomplete or corrupted log entries that are missing key details are removed from the log.
5. The Target Group ARN and Request fields are processed to extract out specific information such as AWS account IDs, AWS region names, URL components, HTTP methods, and HTTP versions.
6. After preprocessing, the rest of the data is transformed into the OCSF HTTP Activity event class, written to a Parquet file from a Pandas DataFrame, and can be further uploaded or processed. (In our case we will use DuckDB for analysis).

The Python script `process_alb.py` does all of the heavy lifting on your behalf, however, it will not attempt to rightsize the amount of data consumed based on your system specifications. If you point the script at a bucket containing several 100 GBs or TBs of ALB access logs, it will not work.

Please consider adapting this script to use PySpark and run it on Databricks or Amazon EMR Serverless Workspaces if you require a large amount of data to be ETL'ed at once.

For the best performance, provide a specific path to a specific month, or a specific day. For instance, consider this example path which only retrieves a single day's worth of logs from a specific account and region: `AWSLogs/12accountexample12/elasticloadbalancing/us-east-2/2025/01/17`

In the next section you will learn about the pertinent parts of the script, or you can optionally skip to the DuckDB Analysis section if you do not care about the inner workings of our (janky) script!



ALB Access Log ETL

The only required steps, beyond installing the prerequisite software dependencies, is to add values for the `BUCKET_NAME` and `PATH_NAME` constants. For the best performance, as previously mentioned, use a path that is specific down to the day.

This information is used by the `openLogFile` method which in turn uses the ListObjectsV2 API to pull out all individual objects (log files) within that prefix, if there are any there. If there are, the files are locally downloaded and gunzipped with each log line sent to the `grokProcessLogs` function.

From there, all of the preprocessing and transformation is carried out by a series of additional chained methods. This is all done serially. To make it faster the script could be further modified to use generators and multi-processing to parallelize multiple preprocessing steps at once.

The `grokProcessLogs` method applies the Grok filter which is instantiated in global space at the beginning of the script.

```
BUCKET_NAME = “my-alb-access-logs”

PATH_NAME = “AWSLogs/123456789012/elasticloadbalancing/us-
east-2/2024/01/17”
```

```
GROK = Grok(

‘%{DATA:type}\s+%{TIMESTAMP_ISO8601:time}\s+%{DATA:elb}\
s+%{DATA:client}\s+%{DATA:target}\s+%{BASE10NUM:request_processing_
time}\s+%{DATA:target_processing_time}\s+%{BASE10NUM:response_
processing_time}\s+%{BASE10NUM:elb_status_code}\s+%{DATA:target_
status_code}\s+%{BASE10NUM:received_bytes}\s+%{BASE10NUM:sent_bytes}\
s+\”%{DATA:request}\”\s+\”%{DATA:user_agent}\”\s+%{DATA:ssl_cipher}\
s+%{DATA:ssl_protocol}\s+%{DATA:target_group_arn}\s+\”%{DATA:trace_
id}\”\s+\”%{DATA:domain_name}\”\s+\”%{DATA:chosen_cert_arn}\”\
s+%{DATA:matched_rule_priority}\s+%{TIMESTAMP_ISO8601:request_
creation_time}\s+\”%{DATA:actions_executed}\”\s+\”%{DATA:redirect_
url}\”\s+\”%{DATA:error_reason}\”\s+\”%{DATA:target_list}\”\
s+\”%{DATA:target_status_code_list}\”\s+\”%{DATA:classification}\”\
s+\”%{DATA:classification_reason}\”’

)
```

Applying the Grok pattern will convert the raw log line into a Python dictionary without further processing. This allows us to control the processing but providing key names for the corresponding log field value.

The dictionary is immediately checked to ensure that it is not incomplete or corrupt by ensuring a target_group_arn field is present. This method is also where the first preprocessing occurs by retrieving the AWS region and account data from the target_group_arn field.

Other processing happens here as well. The "base event" attributes within the HTTP Activity event class are mapped within the `httpActivityBaseEventMapping` method. The `dst_endpoint` OCSF object–of the Network Endpoint object type–is processed using the `elbTargetProcessor` method.

```
def grokProcessLogs(rawlog: str) -> dict | None:
    """
    Uses PyGrok to transform ALB access log pattern into Python dictionary and
further OCSF conversion
    """
    preProcessedLog = GROK.match(rawlog)
    # ALB access log docs don't account for this, but if the TG ARN is empty it is
likely a log delivery error and should be ignored
    try:
        if preProcessedLog["target_group_arn"] is not None and
preProcessedLog["target_group_arn"] != "-":
            try:
                tgSplitter = preProcessedLog["target_group_arn"].split(":")
                preProcessedLog["region"] = tgSplitter[3]
                preProcessedLog["account"] = str(tgSplitter[4])
                baseEventMapping =
httpActivityBaseEventMapping(preProcessedLog["request"].split(" ")[0])
                dstEndpoint = elbTargetProcessor(preProcessedLog["target"])
                ocsf = httpActivityOcsfBuilder(rawlog, preProcessedLog,
baseEventMapping, dstEndpoint)
                return ocsf
            except IndexError:
                pass
    except TypeError:
        return None
```


Finally, the final OCSF HTTP Activity event class is assembled (with other helper functions) in the `httpActivityOcsfBuilder` method.

In the next section you will use DuckDB to run very basic SQL queries against the Parquet file to learn how to conduct simple investigations or exploratory data analysis on ALB logs.


```
{
   "activity_id": 3,
   "activity_name": "Get",
   "category_name": "Network Activity",
   "category_uid": 4,
   "class_name": "HTTP Activity",
   "class_uid": 4002,
   "severity_id": 1,
   "severity": "Informational",
   "status": "Failure",
   "status_code": "502",
   "status_detail": "LambdaInvalidResponse",
   "status_id": 2,
   "type_uid": 400203,
   "type_name": "HTTP Activity: Get",
   "message": "ALB executed the following actions: waf,forward",
   "time": "2024-11-16 23:59:49.000",
   "start_time": "2024-11-16 23:59:49.000",
   "duration": 0.046,
   "raw_data": "http 2024-11-16T23:59:49.445081Z app/mock-lambda-external-
alb/72b69973b95c6ccf 5.8.11.202:60000 - 0.012 0.034 0.000 502 - 96 272
\"GET http://mock-lambda-external-alb-12345678.us-east-2.elb.amazonaws.
com:80/ HTTP/1.0\" \"Vodafone/1.0/V802SE/SEJ001 Browser/SEMC-Browser/4.1\" - -
arn:aws:elasticloadbalancing:us-east-2:123456789012:targetgroup/mock-lambda-
tg/123redactedlol123 \"Root=1-673931f5-3c6b885a0d99945f6cdc880d\" \"-\" \"-\" 0
2024-11-16T23:59:49.399000Z \"waf,forward\" \"-\" \"LambdaInvalidResponse\" \"-\" \"-\"
\"-\" \"-\" TID_5c70dac1416ce84c9b5eee0d785cf295\n",

# ...view the full code block here
```


Analyze ALB Access Logs With DuckDB

In this section you will learn a handful of SQL queries to run against your HTTP Activity OCSF-formatted ALB Access Logs. For a more thorough understanding of DuckDB, refer to the Introduction to DuckDB for SecOps blog we linked at the beginning of this paper.

For a much better security-relevant analysis, you should consider creating (Materialized) Views or using JOINS to put together related data sources. For instance, you can create a View in AWS Glue against VPC Flow Logs, WAF Logs, and ALB Access Logs added as a customer source to Amazon Security Lake to analyze all of the relevant parts of the logs at once.

This can be helpful for troubleshooting network connectivity issues as well as post-blast analysis if SQL Injection or Cross-Site Scripting attack evaded your WAF defenses and routed to a target downstream.

Use the `duckdb_alb_ocsf.py` notebook-ized script within the VSCode Jupyter plugin, and change the `LOCAL_PARQUET` constant to your Parquet file if you changed the name of that in the previous script.

If you're unfamiliar with the dataset, the first action you can take is determining the total amount of rows using the `COUNT(*)` function within SQL. This will print out how many rows are in the normalized dataset.

```
duckdb.sql(
   f"""
   SELECT COUNT(*) FROM read_parquet('{LOCAL_
PARQUET}')
   """
).show()
```

Furthermore, the most perfunctory step to take in exploratory data analysis (EDA) besides getting a row count is taking a sampling of logs. In this case, the column limit of DuckDB is an issue, given the hierarchical nature of OCSF.

Use the `LIMIT` clause within SQL to set the maximum amount of rows returned to avoid overwhelming yourself.

```
duckdb.sql(
   f"""
   SELECT * FROM read_parquet('{LOCAL_PARQUET}')
   LIMIT 30
   """
).show()
```

Using `SELECT DISTINCT` along with specifying certain columns (OCSF attributes) is helpful to get unique instances of certain data points, such as both the source and destination IP addresses. You can use dot-notation to retrieve deeply nested data such as `src_endpoint.ip` or `http_rquest.url.query_string`.

```
duckdb.sql(
   f"""
   SELECT DISTINCT
        src_endpoint.ip,
        dst_endpoint.uid
   FROM read_parquet('{LOCAL_PARQUET}')
   LIMIT 50
   """
).show()
```

Use summary aggregations to get a total count of HTTP Methods along with the specific HTTP Codes and Methods. A summary aggregation is created by using the `COUNT()` operator along with specifying other fields of value for your aggregation.

You then reference those same fields (directly, or via alias) using the `GROUP BY` statement. You can further order these with the `ORDER BY` function, in this case the aggregation method is returned in descending order by the total number of methods.

```
duckdb.sql(
    f"""
    SELECT
        COUNT(activity_name) AS total_methods,
        status_code,
        activity_name as http_method
    FROM read_parquet('{LOCAL_PARQUET}')
    GROUP BY http_method, status_code
    ORDER BY total_methods DESC
    """
).show()
```

total_methods int64	status_code varchar	http_method varchar
386	502	Get
275	403	Get
62	502	Post
13	403	Head
6	502	Options
3	460	Head
2	413	Get
1	200	Post
1	400	Post
1	200	Get

As an example, locally ran against my own sample data returned this resulting table.

In the event there is a HTTP status code of interest, you can pull out specific source and destination information for it. In this case, retrieving the source and destination IP addresses and ports, alongside the user agent, status detail, and message from the OCSF data.

```
duckdb.sql(
    f"""
    SELECT
        src_endpoint.ip as src_ip,
        src_endpoint.port as src_port,
        dst_endpoint.ip as dst_ip,
        dst_endpoint.port as dst_port,
        http_request.user_agent as user_agent,
        message,
        status_detail
    FROM read_parquet('{LOCAL_PARQUET}')
    WHERE status_code = 460
    """
).show()
```

You can use predicates written against specific parts of the normalized schema to find areas where there is not a null query in the URL.

You can further specify this query with other predicates such as a status code. This is useful if you were looking for specific tradecraft such as Blind SQL injection or XSS attempts where the vector is in the URL query string.

You can further filter this down with additional predicates to search for successful attempts or attempts from specific IP addresses or against a specific URL in your application stack.

```
duckdb.sql(
    f"""
    SELECT
        activity_name as http_method,
        status,
        src_endpoint.ip,
        http_request.url.query_string as query_string
    FROM read_parquet('{LOCAL_PARQUET}')
    WHERE http_request.url.query_string IS NOT NULL
    """
).show()
```

There are several other types of analysis that you could conduct, ultimately it comes down to your own reporting and security investigatory requirements.

Wrap Up

Security data is everywhere and the ALB access log is just one of many kinds of data we're interested in as security professionals. While this paper went over the specifics of parsing ALB access logs, mapping them to OCSF, and storing and analyzing them, you will find the general concepts can be applied to other types of security-relevant data, as well.

Whatever data you start with, keep your eye on where you're going. Performing ETL on a large amount of data is not trivial. You should always work backwards from specific use cases generated from appropriate threat modeling and security data metrics workshops conducted in your own company or by the Query SecDataOps specialist team.

For more information, request a meeting with our sales team and see if you're a good fit for a SecDataOps Workshop!

Query: Making Open Federated Search for Security a Reality

Query aims to deliver visibility into all relevant data for security teams. We provide a **federated search solution** that allows operators to **access data at the source** and in your data lakes, creating opportunities for more nimble and cost efficient data storage architectures.

Our customers are using Query to expand visibility for security investigations, threat hunting, and incident response. They are drastically **reducing the time and complexity** of repetitive search tasks and **improving outcomes for investigations.** Expose your security data with Query.

Ready to expedite your security investigations with open federated search for security?

For more information visit: www.query.ai