AWS Security Lake Questions

What is AWS Security Lake, and what is its primary purpose in a security operations environment?

AWS Security Lake is a purpose-built data lake service that automatically centralizes security data from AWS environments, SaaS providers, on-premises sources, and cloud sources into a customer-owned data lake stored in Amazon S3. It normalizes data into the Open Cybersecurity Schema Framework (OCSF) format, enabling comprehensive security analytics and threat detection.

Primary purposes:

Centralized security data repository: **Multi-source aggregation** - automatically collects data from AWS services (CloudTrail, VPC Flow Logs, Route 53 query logs, Security Hub findings, etc.), SaaS applications (Okta, Salesforce, Crowdstrike, etc.), third-party security tools, custom sources via API. **Scale** - handles petabytes of security data, S3-based storage provides virtually unlimited capacity, cost-effective long-term retention. **Data lake architecture** - raw data preserved for forensics, partitioned by source and time for efficient querying, supports both real-time and historical analysis.

Data normalization and standardization: **OCSF format** - Open Cybersecurity Schema Framework providing common schema, converts disparate log formats into unified structure, maintains source fidelity while enabling cross-source correlation. **Benefits** - write queries once, work across all data sources, easier to build detection rules, simplifies tool integration and data sharing. **Example**: AWS CloudTrail, Okta logs, and firewall logs all normalized to same schema making cross-environment threat hunting possible with single query.

Security analytics enablement: **Native AWS integration** - direct integration with Amazon Athena for SQL queries, Amazon QuickSight for visualization, Amazon SageMaker for ML-based threat detection, Amazon OpenSearch for real-time analytics. **Third-party SIEM integration** - connectors for Splunk, Datadog, Sumo Logic, custom SIEMs, enables "bring your own analytics". **Cost optimization** - S3 storage significantly cheaper than traditional SIEM storage, tiered storage (S3 Standard, Infrequent Access, Glacier) for retention, only pay for what you query.

Key architectural components:

Data sources: **AWS native sources** - CloudTrail management and data events, VPC Flow Logs, Route 53 resolver query logs, Security Hub findings, AWS WAF logs, Lambda execution logs, EKS audit logs, S3 access logs. **Custom sources** - AWS SDK/API for ingestion, AWS Lambda for transformation, AWS Glue for ETL, direct S3 upload with metadata.

Data lake structure:

```
S3 Bucket: aws-security-data-lake-{region}-{account}
```

```
├──── aws_cloudtrail/
│    ├──── region=us-east-1/
│    │    ├──── accountId=123456789012/
│    │    │    ├──── date=2024-01-20/
│    │    │    │    ├──── hour=00/
│    │    │    │    │    └──── data.parquet
│    │    │    │    ├──── hour=01/
│    │    │    │    │    └──── data.parquet
├──── vpc_flow/
│    ├──── region=us-west-2/
│    │    ├──── accountId=123456789012/
│    │    │    ├──── date=2024-01-20/
├──── route53/
├──── security_hub/
└──── custom_sources/
    ├──── okta_logs/
    ├──── crowdstrike_detections/
    └──── palo_alto_firewall/
```

OCSF normalization: **Common fields across all sources** - timestamp, severity, category, type_uid (event type), metadata (version, product, etc.), actor (who), resource (what), observables (IOCs). **Source-specific extensions** - preserves original fields in unmapped section, maintains forensic value, allows source-specific queries when needed.

Example OCSF event:

```
{
  "time": "2024-01-20T10:30:00Z",
  "severity_id": 3,
  "class_uid": 3002,
  "class_name": "Authentication",
  "type_uid": 300201,
  "type_name": "Authentication: Logon",
  "category_uid": 3,
  "category_name": "Identity & Access Management",
  "activity_id": 1,
  "activity_name": "Logon",
  "actor": {
    "user": {
      "name": "alice@example.com",
      "uid": "arn:aws:iam::123456789012:user/alice"
    },
    "session": {
      "uid": "session-abc-123"
    }
  },
  "device": {
    "ip": "203.0.113.45",
    "location": {
      "country": "US",
      "region": "CA"
    }
  },
  "cloud": {
    "provider": "AWS",
    "region": "us-east-1"
  },
  "observables": [
    {
      "name": "src_endpoint.ip",
      "type_id": 2,
      "value": "203.0.113.45"
    }
  ],
  "metadata": {
    "product": {
      "name": "CloudTrail",
      "vendor_name": "AWS"
    },
    "version": "1.0.0"
  },
  "unmapped": {
    // Original CloudTrail fields not in OCSF
    "userAgent": "aws-cli/2.13.0",
    "requestParameters": {...}
  }
}
```

From security engineering perspective:

Architecture benefits: **Decoupled storage and compute** - data stored once, query with multiple tools, no vendor lock-in for analytics, data portability. **Cost efficiency** - S3 storage ~$0.023/GB/month vs. traditional SIEM $100-300/GB/month, query costs only when analyzing, retention measured in years not days. **Scalability** - designed for cloud-scale security data, handles spiky ingestion patterns, queries across petabytes efficient with partitioning.

Security use cases: **Threat detection** - real-time alerting via Amazon EventBridge, historical threat hunting across months/years of data, correlation across AWS and third-party sources. **Compliance** - centralized audit trail for all environments, long-term retention meeting regulatory requirements (7+ years), immutable storage with S3 Object Lock. **Incident response** - comprehensive forensic data available immediately, timeline reconstruction across all sources, retain evidence for legal proceedings. **Threat hunting** - proactive searches for IOCs across entire estate, behavioral analysis identifying anomalies, ML-based pattern detection.

Example deployment scenario: Enterprise with multi-account AWS Organization, Okta for SSO, Palo Alto firewalls would: enable Security Lake in security account (delegated administrator), automatically collect CloudTrail, VPC Flow, Route 53 from all accounts, ingest Okta authentication logs via custom source, ingest firewall logs via AWS Lambda, configure Athena for SQL-based hunting, set up EventBridge rules for real-time alerting, integrate with existing SIEM for operational workflows, and retain data for 7 years meeting compliance requirements.

Cost comparison (100 TB security data):

```
Traditional SIEM:
- Ingestion: 100 TB × $150/GB = $15M/year
- Storage: Limited retention (90 days typical)

Security Lake:
- Ingestion: Included (AWS native sources)
- Storage: 100 TB × $0.023/GB × 12 = $27,600/year
- Query: ~$5,000/month (Athena) = $60,000/year
- Total: ~$88,000/year (99.4% cost reduction)
```

Security Lake represents paradigm shift from expensive, limited-retention SIEMs to cost-effective, unlimited-scale security data platform enabling comprehensive visibility, long-term retention, and flexible analytics without vendor lock-in.

What is the Open Cybersecurity Schema Framework (OCSF), and why is it important for Security Lake?

OCSF is an open-source framework providing a vendor-agnostic, extensible schema for security telemetry data. It's critical to Security Lake's value proposition, enabling unified analytics across diverse security data sources.

OCSF fundamentals:

What it solves: **Data heterogeneity problem** - every security tool has different log format (CloudTrail JSON vs. syslog vs. CEF vs. proprietary), same event described differently across sources, makes cross-source correlation extremely difficult. **Example**: User authentication described as:

```
CloudTrail: {"eventName": "ConsoleLogin", "userIdentity": {...}}
Okta: {"eventType": "user.session.start", "actor": {...}}
Azure AD: {"operationName": "Sign-in activity", "userPrincipalName": "..."}
```

OCSF normalization: All map to common schema:

```
{
  "class_name": "Authentication",
  "type_name": "Authentication: Logon",
  "actor": {
    "user": {
      "name": "alice@example.com"
    }
  }
}
```

Core OCSF concepts:

Event classes - categories of security events:

- **System Activity** (1xxx): Process activity, file activity, kernel activity, module activity
- **Findings** (2xxx): Detection finding, vulnerability finding, compliance finding
- **Identity & Access Management** (3xxx): Authentication, authorization, entity management
- **Network Activity** (4xxx): Network connection, HTTP activity, DNS activity, DHCP activity
- **Discovery** (5xxx): Device inventory discovery, user inventory
- **Application Activity** (6xxx): Web resource access, API activity

Example event class structure:

```
// Authentication class (3002)
{
  "class_uid": 3002,
  "class_name": "Authentication",
  "type_uid": 300201,  // Logon
  "type_name": "Authentication: Logon",
  "activity_id": 1,
  "activity_name": "Logon",

  // Required attributes
  "time": "2024-01-20T10:30:00Z",
  "severity_id": 1,

  // Recommended attributes
  "actor": {
    "user": {
      "name": "alice@example.com",
      "uid": "user-123",
      "type_id": 1  // User
    },
    "session": {
      "uid": "session-abc"
    }
  },

  "device": {
    "ip": "203.0.113.45",
    "hostname": "workstation-01"
  },

  "dst_endpoint": {
    "ip": "10.0.1.50",
    "port": 443
  },

  "status_id": 1,  // Success
  "status": "Success",

  // Optional attributes
  "auth_protocol_id": 1,  // NTLM, Kerberos, etc.
  "logon_type_id": 2,  // Interactive

  // Observables for IOC extraction
  "observables": [
    {
      "name": "actor.user.name",
      "type_id": 4,  // Username
      "value": "alice@example.com"
    },
    {
      "name": "device.ip",
      "type_id": 2,  // IP Address
      "value": "203.0.113.45"
    }
  ],

  // Metadata
  "metadata": {
    "product": {
      "name": "CloudTrail",
      "vendor_name": "AWS",
      "version": "1.0"
    },
    "version": "1.1.0",
    "logged_time": "2024-01-20T10:30:01Z"
  },

  // Unmapped preserves source data
  "unmapped": {
    "requestParameters": {
      "mfaAuthenticated": "true"
    },
    "userAgent": "AWS-Console"
  }
}
```

Why OCSF matters for Security Lake:

1. Cross-source analytics: **Single query across all sources**:

```
-- Find all failed authentications across ALL sources
-- (CloudTrail, Okta, AD, firewalls, etc.)
SELECT
```

```
    time,
    actor.user.name AS user,
    device.ip AS source_ip,
    metadata.product.name AS source,
    status AS result
FROM security_lake_database.ocsf_authentication
WHERE
    date BETWEEN '2024-01-01' AND '2024-01-31'
    AND status_id != 1  -- Not successful
ORDER BY time DESC
```

Without OCSF, would need separate queries per source:

```
-- CloudTrail
SELECT ... FROM cloudtrail WHERE eventName = 'ConsoleLogin' AND errorCode IS NOT NULL

-- Okta
SELECT ... FROM okta WHERE eventType LIKE '%failure%'

-- AD
SELECT ... FROM active_directory WHERE EventID = 4625

-- Then manually correlate results
```

2. Detection rule portability: Write once, works everywhere:

```
-- Detect brute force across ANY authentication source
SELECT
    actor.user.name,
    device.ip,
    COUNT(*) AS failed_attempts,
    metadata.product.name AS sources
FROM ocsf_authentication
WHERE
    date = CURRENT_DATE
    AND status_id != 1
GROUP BY
    actor.user.name,
    device.ip,
    metadata.product.name
HAVING COUNT(*) > 50
```

3. Tool interoperability: **SIEM integration simplified** - Splunk, Datadog, Sumo Logic all understand OCSF, write integration once for OCSF, works with all Security Lake data. **Example Splunk search**:

```
index=security_lake class_name="Authentication" status_id!=1
```

```
| stats count by actor.user.name, device.ip
| where count > 50
```

4. Threat intelligence correlation: **Standard observable extraction**:

```
-- Find IOCs from threat feed in ANY data source
WITH ThreatIOCs AS (
    SELECT ioc_value, ioc_type
    FROM threat_intelligence
    WHERE last_seen > CURRENT_DATE - INTERVAL '7' DAY
)
SELECT DISTINCT
    o.value AS matched_ioc,
    t.ioc_type,
    e.class_name,
    e.time,
    e.actor.user.name,
    e.metadata.product.name AS source
FROM security_lake_database.all_events e
CROSS JOIN UNNEST(e.observables) AS o
JOIN ThreatIOCs t ON o.value = t.ioc_value
WHERE e.date >= CURRENT_DATE - INTERVAL '1' DAY
```

OCSF mapping examples:

CloudTrail → OCSF:

```
# Mapping logic (simplified)
def cloudtrail_to_ocsf(cloudtrail_event):
    return {
        "class_uid": 3002,  # Authentication
        "type_uid": 300201,  # Logon
        "time": cloudtrail_event["eventTime"],
        "actor": {
            "user": {
                "name": cloudtrail_event["userIdentity"]["userName"],
                "uid": cloudtrail_event["userIdentity"]["arn"],
                "type_id": 1  # User
            }
        },
        "device": {
            "ip": cloudtrail_event["sourceIPAddress"]
        },
        "cloud": {
            "provider": "AWS",
            "region": cloudtrail_event["awsRegion"],
            "account": {
                "uid": cloudtrail_event["userIdentity"]["accountId"]
            }
        },
        "status_id": 1 if cloudtrail_event.get("errorCode") is None else 2,
        "metadata": {
            "product": {
                "name": "CloudTrail",
                "vendor_name": "AWS"
            },
            "version": "1.1.0"
        },
        "unmapped": {
            # Preserve CloudTrail-specific fields
            "eventName": cloudtrail_event["eventName"],
            "requestParameters": cloudtrail_event.get("requestParameters"),
            "userAgent": cloudtrail_event.get("userAgent")
        }
    }
```

VPC Flow Logs → OCSF:

```
def vpc_flow_to_ocsf(flow_log):
    return {
        "class_uid": 4001,  # Network Activity
        "type_uid": 400101,  # Network Connection
        "time": datetime.fromtimestamp(flow_log["start"]).isoformat(),
        "src_endpoint": {
            "ip": flow_log["srcaddr"],
            "port": flow_log["srcport"]
        },
        "dst_endpoint": {
            "ip": flow_log["dstaddr"],
            "port": flow_log["dstport"]
        },
        "connection_info": {
            "protocol_num": flow_log["protocol"],
            "protocol_name": get_protocol_name(flow_log["protocol"]),
            "direction_id": 1 if flow_log["direction"] == "ingress" else 2
        },
        "traffic": {
            "bytes": flow_log["bytes"],
            "packets": flow_log["packets"]
        },
        "status_id": 1 if flow_log["action"] == "ACCEPT" else 2,
        "cloud": {
            "provider": "AWS",
            "region": flow_log["region"]
        },
        "metadata": {
            "product": {
                "name": "VPC Flow Logs",
                "vendor_name": "AWS"
            },
            "version": "1.1.0"
        },
        "observables": [
            {"name": "src_endpoint.ip", "type_id": 2, "value": flow_log["srcaddr"]},
            {"name": "dst_endpoint.ip", "type_id": 2, "value": flow_log["dstaddr"]}
        ]
    }
```

Custom source → OCSF (Palo Alto firewall):

```
def palo_alto_to_ocsf(pa_log):
    return {
        "class_uid": 4002,  # Network Activity: HTTP
        "type_uid": 400201,  # HTTP Activity
        "time": pa_log["receive_time"],
        "src_endpoint": {
            "ip": pa_log["src"],
            "port": pa_log["sport"],
            "zone": pa_log["from"]
        },
        "dst_endpoint": {
            "ip": pa_log["dst"],
            "port": pa_log["dport"],
            "zone": pa_log["to"]
        },
        "http_request": {
            "url": {
                "hostname": pa_log["misc"],
                "path": pa_log["url"]
            },
            "http_method": pa_log["http_method"],
            "user_agent": pa_log["user_agent"]
        },
        "firewall_rule": {
            "uid": pa_log["rule"],
            "name": pa_log["rule"]
        },
        "disposition_id": 1 if pa_log["action"] == "allow" else 2,
        "severity_id": get_severity(pa_log["threat_category"]),
        "metadata": {
            "product": {
                "name": "Palo Alto Firewall",
                "vendor_name": "Palo Alto Networks"
            },
            "version": "1.1.0"
        },
        "unmapped": {
            # PA-specific fields
            "threat_id": pa_log["threatid"],
            "category": pa_log["category"],
            "pcap_id": pa_log["pcap_id"]
        }
    }
```

Benefits in practice:

Scenario 1: Multi-source threat hunt:

```
-- Hunt for lateral movement across AWS, Okta, and on-prem AD
-- Single query works because all normalized to OCSF

WITH AuthEvents AS (
    SELECT
        time,
        actor.user.name AS user,
        device.ip AS source_ip,
        dst_endpoint.ip AS target,
        metadata.product.name AS auth_source
    FROM ocsf_authentication
    WHERE date >= CURRENT_DATE - INTERVAL '24' HOUR
    AND status_id = 1  -- Successful
)
SELECT
    user,
    COUNT(DISTINCT source_ip) AS unique_sources,
    COUNT(DISTINCT target) AS unique_targets,
    COUNT(DISTINCT auth_source) AS data_sources,
    ARRAY_AGG(DISTINCT auth_source) AS sources_list
FROM AuthEvents
GROUP BY user
HAVING
    COUNT(DISTINCT source_ip) > 5
    AND COUNT(DISTINCT target) > 3
ORDER BY unique_targets DESC
```

Scenario 2: Compliance reporting:

```
-- PCI-DSS Requirement 10.2: Audit privileged access
-- Works across all systems without custom parsing

SELECT
    DATE_TRUNC('day', time) AS day,
    actor.user.name,
    COUNT(*) AS privileged_actions,
    ARRAY_AGG(DISTINCT metadata.product.name) AS systems_accessed
FROM security_lake_database.all_events
WHERE
    date >= CURRENT_DATE - INTERVAL '90' DAY
    AND (
        -- Privileged indicators in OCSF
        actor.user.type_id = 2  -- Admin user
        OR class_name IN ('Authorization', 'Entity Management')
        OR severity_id >= 3  -- Medium or higher
    )
GROUP BY
    DATE_TRUNC('day', time),
    actor.user.name
ORDER BY day DESC, privileged_actions DESC
```

OCSF extensibility: **Custom attributes** - add organization-specific fields while maintaining compatibility:

```
{
  "class_name": "Authentication",
  // ... standard OCSF fields ...
  "custom": {
    "risk_score": 85,
    "business_unit": "Finance",
    "data_classification": "Confidential",
    "compliance_scope": ["PCI", "SOX"]
  }
}
```

Limitations and considerations: **Not all fields map perfectly** - some source-specific data goes to `unmapped`, important forensic details might be in unmapped, need to know when to reference unmapped. **Schema evolution** - OCSF updates periodically, need version management strategy, backward compatibility considerations. **Transformation overhead** - normalization adds processing latency (seconds), acceptable for most use cases, critical for real-time alerting considerations.

Best practices: Use OCSF fields for detection rules (portability), reference `unmapped` for source-specific investigations, include `metadata.product.name` in queries to track sources, leverage `observables` array for IOC extraction, stay current with OCSF schema versions, contribute mappings back to OCSF community, document custom extensions clearly.

OCSF is fundamental to Security Lake's value - it transforms disparate security data into unified, queryable format enabling cross-source analytics, portable detection logic, and tool interoperability impossible with proprietary schemas. Understanding OCSF schema is essential for effective Security Lake usage.

How do you ingest custom data sources into AWS Security Lake, and what are the best practices for data transformation?

Custom source ingestion enables Security Lake to consolidate data beyond AWS native sources, creating comprehensive security data repository including third-party tools, SaaS applications, and on-premises systems.

Custom source ingestion methods:

Method 1: Direct S3 upload with OCSF conversion (recommended for batch data):

Architecture:

```
Data Source → Lambda (Transform to OCSF) → S3 Security Lake Bucket
                     ↓
           Glue Catalog (Metadata)
```

Implementation example (Okta logs):

```
import json
import boto3
import pyarrow as pa
import pyarrow.parquet as pq
from datetime import datetime
import hashlib

s3_client = boto3.client('s3')
glue_client = boto3.client('glue')

SECURITY_LAKE_BUCKET = "aws-security-data-lake-us-east-1-123456789012"
CUSTOM_SOURCE_PREFIX = "ext/okta_logs"

def lambda_handler(event, context):
    """
    Transform Okta logs to OCSF and upload to Security Lake
    Triggered by: EventBridge schedule or S3 upload to staging bucket
    """
    # Get Okta logs (from API, S3 staging, etc.)
    okta_logs = fetch_okta_logs()

    # Transform to OCSF
    ocsf_events = [transform_okta_to_ocsf(log) for log in okta_logs]

    # Convert to Parquet (Security Lake requirement)
    parquet_data = convert_to_parquet(ocsf_events)
    # Upload with proper partitioning
    upload_to_security_lake(parquet_data, ocsf_events[0]['time'])

    # Update Glue Catalog
    update_glue_catalog()

    return {
        'statusCode': 200,
        'body': f'Processed {len(ocsf_events)} events'
    }

def transform_okta_to_ocsf(okta_event):
    """Transform Okta authentication event to OCSF"""
    return {
        "class_uid": 3002,  # Authentication
        "class_name": "Authentication",
        "type_uid": get_auth_type_uid(okta_event['eventType']),
        "type_name": f"Authentication: {okta_event['eventType']}",
        "time": okta_event['published'],
        "severity_id": map_okta_severity(okta_event['severity']),

        "actor": {
            "user": {
                "name": okta_event['actor']['alternateId'],
                "uid": okta_event['actor']['id'],
                "type_id": 1,  # User
                "email_addr": okta_event['actor']['alternateId']
            },
            "session": {
                "uid": okta_event.get('authenticationContext',
{}).get('externalSessionId')
            }
        },

        "device": {
            "ip": okta_event['client']['ipAddress'],
            "location": {
                "city": okta_event['client'].get('geographicalContext',
{}).get('city'),
                "country": okta_event['client'].get('geographicalContext',
{}).get('country'),
                "coordinates": {
                    "lat": okta_event['client'].get('geographicalContext',
{}).get('geolocation', {}).get('lat'),
                    "lon": okta_event['client'].get('geographicalContext',
{}).get('geolocation', {}).get('lon')
                }
            },
            "os": {
                "name": okta_event['client']['userAgent']['os'],
                "version": okta_event['client']['userAgent']['osVersion']
            },
            "browser": {
                "name": okta_event['client']['userAgent']['browser'],
                "version": okta_event['client']['userAgent']['browserVersion']
            }
        },

        "dst_endpoint": {
            "application": {
                "name": okta_event['target'][0]['displayName'] if
okta_event.get('target') else None
            }
        },

        "status_id": 1 if okta_event['outcome']['result'] == 'SUCCESS' else 2,
        "status": okta_event['outcome']['result'],

        "auth_protocol": okta_event.get('authenticationContext',
{}).get('credentialProvider'),

        "observables": [
            {
                "name": "actor.user.email_addr",
                "type_id": 4,  # Email
                "value": okta_event['actor']['alternateId']
            },
            {
                "name": "device.ip",
                "type_id": 2,  # IP Address
                "value": okta_event['client']['ipAddress']
            }
        ],

        "metadata": {
            "product": {
                "name": "Okta",
                "vendor_name": "Okta",
                "version": okta_event['version']
            },
            "version": "1.1.0",  # OCSF version
            "logged_time": okta_event['published'],
            "uid": okta_event['uuid']
        },

        # Preserve Okta-specific fields
        "unmapped": {
            "transaction_id": okta_event['transaction']['id'],
            "request_id": okta_event['debugContext']['debugData']['requestId'],
            "authentication_step": okta_event.get('authenticationContext',
{}).get('authenticationStep'),
            "risk_level": okta_event.get('securityContext', {}).get('riskLevel'),
            "original_event": json.dumps(okta_event)  # Full original for forensics
        }
    }

def convert_to_parquet(ocsf_events):
    """Convert OCSF JSON to Parquet format"""
    # Define schema
    schema = pa.schema([
        ('class_uid', pa.int32()),
        ('class_name', pa.string()),
        ('time', pa.timestamp('us')),
        ('severity_id', pa.int32()),
        ('actor', pa.struct([
            ('user', pa.struct([
                ('name', pa.string()),
                ('uid', pa.string()),
                ('email_addr', pa.string())
            ]))
        ])),
        # ... additional fields
    ])

    # Convert to PyArrow table
    table = pa.Table.from_pylist(ocsf_events, schema=schema)

    # Write to Parquet bytes
    buf = pa.BufferOutputStream()
    pq.write_table(table, buf, compression='SNAPPY')

    return buf.getvalue().to_pybytes()

def upload_to_security_lake(parquet_data, event_time):
    """
    Upload to Security Lake with proper partitioning
    Partition structure: region/accountId/eventDay/eventHour/
    """
    event_dt = datetime.fromisoformat(event_time.replace('Z', '+00:00'))

    # Security Lake partition structure
    partition_path = (
        f"{CUSTOM_SOURCE_PREFIX}/"
        f"region=global/"
        f"accountId=okta/"
        f"eventDay={event_dt.strftime('%Y%m%d')}/"
        f"eventHour={event_dt.strftime('%H')}/"
    )

    # Generate unique filename
    file_hash = hashlib.md5(parquet_data).hexdigest()[:8]
    filename = f"data-{event_dt.strftime('%Y%m%d-%H%M%S')}-{file_hash}.parquet"
    s3_key = f"{partition_path}{filename}"

    # Upload to S3
    s3_client.put_object(
        Bucket=SECURITY_LAKE_BUCKET,
        Key=s3_key,
        Body=parquet_data,
        ServerSideEncryption='aws:kms',
        Metadata={
            'source': 'okta',
            'ocsf-version': '1.1.0',
            'event-count': str(len(json.loads(parquet_data)))
        }
    )

    return s3_key

def update_glue_catalog():
    """Register custom source in Glue Catalog for Athena queries"""
    try:
        glue_client.create_table(
            DatabaseName='security_lake_database',
            TableInput={
                'Name': 'okta_authentication',
                'StorageDescriptor': {
                    'Columns': [
                        {'Name': 'class_uid', 'Type': 'int'},
                        {'Name': 'class_name', 'Type': 'string'},
                        {'Name': 'time', 'Type': 'timestamp'},
                        {'Name': 'actor', 'Type':
'struct<user:struct<name:string,uid:string>>'},
                        # ... additional columns
                    ],
                    'Location':
f's3://{SECURITY_LAKE_BUCKET}/{CUSTOM_SOURCE_PREFIX}/',
                    'InputFormat':
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat',
                    'OutputFormat':
'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat',
                    'SerdeInfo': {
                        'SerializationLibrary':
'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
                    }
                },
                'PartitionKeys': [
                    {'Name': 'region', 'Type': 'string'},
                    {'Name': 'accountId', 'Type': 'string'},
                    {'Name': 'eventDay', 'Type': 'string'},
                    {'Name': 'eventHour', 'Type': 'string'}
                ],
                'TableType': 'EXTERNAL_TABLE'
            }
        )
    except glue_client.exceptions.AlreadyExistsException:
        # Table already exists
        pass
```

Method 2: Custom Source API (for real-time streaming):

Architecture:

```
Data Source → Kinesis Data Firehose → Lambda (Transform) → S3 Security Lake
                                          ↓
                                  CloudWatch Logs (Errors)
```

Firehose transformation Lambda:

```
import base64
import json

def lambda_handler(event, context):
    """
    Kinesis Firehose transformation function
    Converts incoming logs to OCSF format
    """
    output_records = []

    for record in event['records']:
        # Decode input
        payload = base64.b64decode(record['data'])
        source_event = json.loads(payload)

        try:
            # Transform to OCSF
            ocsf_event = transform_to_ocsf(source_event)

            # Re-encode
            output_data = json.dumps(ocsf_event) + '\n'
            output_record = {
                'recordId': record['recordId'],
                'result': 'Ok',
                'data': base64.b64encode(output_data.encode()).decode()
            }
        except Exception as e:
            # Mark as failed for retry
            print(f"Transformation error: {str(e)}")
            output_record = {
                'recordId': record['recordId'],
                'result': 'ProcessingFailed',
                'data': record['data']  # Return original
            }

        output_records.append(output_record)

    return {'records': output_records}
```

Method 3: AWS Glue ETL Job (for large-scale batch processing):

Glue job script:

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue.dynamicframe import DynamicFrame
from pyspark.sql.functions import *
from pyspark.sql.types import *

args = getResolvedOptions(sys.argv, ['JOB_NAME', 'source_bucket',
'security_lake_bucket'])

sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

# Read source data
source_df = spark.read.json(f"s3://{args['source_bucket']}/firewall-logs/")

# Define transformation UDF
def transform_firewall_to_ocsf(row):
    return {
        "class_uid": 4001,  # Network Activity
        "time": row['timestamp'],
        "src_endpoint": {
            "ip": row['source_ip'],
            "port": row['source_port']
        },
        "dst_endpoint": {
            "ip": row['dest_ip'],
            "port": row['dest_port']
        },
        "disposition_id": 1 if row['action'] == 'allow' else 2,
        "metadata": {
            "product": {
                "name": "Palo Alto Firewall"
            }
        }
    }

# Apply transformation
ocsf_df = source_df.rdd.map(transform_firewall_to_ocsf).toDF()

# Write to Security Lake with partitioning
ocsf_df.write \
    .partitionBy("eventDay", "eventHour") \
    .parquet(f"s3://{args['security_lake_bucket']}/ext/firewall/", mode="append")

job.commit()
```

Best practices for custom source ingestion:

1. Data transformation:

- Map to appropriate OCSF class (don't force fit)
- Preserve original data in `unmapped` field for forensics
- Include comprehensive `observables` for IOC extraction
- Validate OCSF schema compliance before upload
- Handle missing fields gracefully (optional vs. required)

2. Partitioning strategy:

```
Required structure:
/region=<region>/
  accountId=<account>/
    eventDay=<YYYYMMDD>/
      eventHour=<HH>/
        data-<timestamp>-<hash>.parquet

Example:
/region=us-east-1/
  accountId=123456789012/
    eventDay=20240120/
      eventHour=14/
        data-20240120-140532-a1b2c3d4.parquet
```

Benefits: Efficient time-range queries, automatic data lifecycle management, supports Athena partition pruning.

3. File format:

- Use Parquet (required by Security Lake)
- Enable Snappy compression
- Optimal row group size: 128MB

- Include metadata in Parquet footer

4. **Data quality**:

```
def validate_ocsf_event(event):
    """Validate OCSF compliance before ingestion"""
    required_fields = ['class_uid', 'class_name', 'time', 'metadata']

    # Check required fields
    for field in required_fields:
        if field not in event:
            raise ValueError(f"Missing required field: {field}")

    # Validate time format
    try:
        datetime.fromisoformat(event['time'].replace('Z', '+00:00'))
    except:
        raise ValueError(f"Invalid time format: {event['time']}")

    # Validate severity_id range
    if 'severity_id' in event and event['severity_id'] not in range(0, 7):
        raise ValueError(f"Invalid severity_id: {event['severity_id']}")

    # Validate observables structure
    if 'observables' in event:
        for obs in event['observables']:
            if 'name' not in obs or 'value' not in obs:
                raise ValueError("Observable missing name or value")

    return True
```

5. **Error handling and monitoring**:

```
import boto3
from botocore.exceptions import ClientError

cloudwatch = boto3.client('cloudwatch')

def ingest_with_monitoring(events):
    """Ingest with CloudWatch metrics"""
    success_count = 0
    failure_count = 0

    for event in events:
        try:
            validate_ocsf_event(event)
            upload_to_security_lake(event)
            success_count += 1
        except Exception as e:
            failure_count += 1
            log_error(event, str(e))

    # Publish metrics
    cloudwatch.put_metric_data(
        Namespace='SecurityLake/CustomIngestion',
        MetricData=[
            {
                'MetricName': 'EventsIngested',
                'Value': success_count,
                'Unit': 'Count'
            },
            {
                'MetricName': 'IngestionFailures',
                'Value': failure_count,
                'Unit': 'Count'
            }
        ]
    )

    # Alert on high failure rate
    if failure_count / len(events) > 0.05:  # >5% failure
        send_alert(f"High ingestion failure rate: {failure_count}/{len(events)}")
```

6. **Cost optimization**:

- Batch events before upload (reduce S3 PUT requests)
- Compress Parquet files (Snappy compression 3-5x)
- Use S3 Intelligent-Tiering for older data
- Implement data lifecycle policies

7. **Security**:

- Encrypt at rest (S3-KMS)
- Encrypt in transit (TLS)
- Use IAM roles (no hardcoded credentials)
- Implement least privilege
- Audit all ingestion activity

Example end-to-end implementation (CrowdStrike detections):

```
# Step 1: Fetch from CrowdStrike API
def fetch_crowdstrike_detections():
    import requests

    api_token = get_secret("crowdstrike-api-token")
    response = requests.get(
        "https://api.crowdstrike.com/detects/queries/detects/v1",
        headers={"Authorization": f"Bearer {api_token}"},
        params={"filter": f"created_timestamp:>'{get_last_ingestion_time()}'"}
    )

    return response.json()['resources']

# Step 2: Transform to OCSF
def transform_crowdstrike_to_ocsf(detection):
    return {
        "class_uid": 2004,  # Detection Finding
        "class_name": "Detection Finding",
        "time": detection['created_timestamp'],
        "severity_id": map_severity(detection['max_severity']),

        "finding_info": {
            "uid": detection['detection_id'],
            "title": detection['behaviors'][0]['tactic'],
            "desc": detection['behaviors'][0]['scenario'],
            "types": [detection['behaviors'][0]['technique']]
        },

        "resources": [{
            "uid": detection['device']['device_id'],
            "name": detection['device']['hostname'],
            "type_id": 6,  # Computer
            "labels": detection['device']['tags']
        }],

        "malware": [{
            "name": detection['behaviors'][0]['display_name'],
            "classification_ids":
[map_malware_classification(detection['behaviors'][0]['objective'])]
        }],

        "observables": extract_iocs(detection),

        "metadata": {
            "product": {
                "name": "CrowdStrike Falcon",
                "vendor_name": "CrowdStrike"
            },
            "version": "1.1.0"
        },

        "unmapped": {
            "confidence": detection['max_confidence'],
            "show_in_ui": detection['show_in_ui'],
            "status": detection['status'],
            "full_detection": json.dumps(detection)
        }
    }

# Step 3: Upload to Security Lake
def ingest_crowdstrike_detections(event, context):
    detections = fetch_crowdstrike_detections()
    ocsf_events = [transform_crowdstrike_to_ocsf(d) for d in detections]

    # Batch and upload
    parquet_data = convert_to_parquet(ocsf_events)
    upload_to_security_lake(parquet_data, ocsf_events[0]['time'])

    # Update last ingestion time
    update_ingestion_checkpoint(ocsf_events[-1]['time'])

    return {'status': 'success', 'events': len(ocsf_events)}
```

Custom source ingestion transforms Security Lake into comprehensive security data platform, but requires careful attention to OCSF mapping, data quality, partitioning, and operational concerns to ensure reliable, performant, cost-effective implementation.

How do you query and analyze data in AWS Security Lake using Amazon Athena, and what are optimization techniques for large-scale queries?

Amazon Athena provides serverless SQL query capability for Security Lake data, enabling interactive analysis, threat hunting, and compliance reporting across petabytes of security telemetry.

Athena fundamentals for Security Lake:

Query basics:

```
-- Security Lake tables automatically created by AWS
-- Format: amazon_security_lake_table_<region>_<source>_<version>

-- Query CloudTrail events
SELECT
    time,
    actor.user.name AS user,
    api.operation AS action,
    cloud.region,
    status
FROM amazon_security_lake_table_us_east_1_cloud_trail_1_0
WHERE
```

```
    eventday = '20240120'
    AND actor.user.name = 'admin@example.com'
ORDER BY time DESC
LIMIT 100;

-- Query VPC Flow Logs
SELECT
    time,
    src_endpoint.ip AS source_ip,
    dst_endpoint.ip AS dest_ip,
    dst_endpoint.port AS dest_port,
    traffic.bytes AS bytes_transferred,
    connection_info.protocol_name AS protocol
FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
WHERE
    eventday = '20240120'
    AND traffic.bytes > 1000000000  -- >1GB
ORDER BY traffic.bytes DESC;
```

Advanced threat hunting queries:

1. Credential access detection:

```
-- Detect password spraying across authentication sources
WITH FailedLogins AS (
    SELECT
        time,
        actor.user.name AS username,
        device.ip AS source_ip,
        metadata.product.name AS auth_source
    FROM amazon_security_lake_table_us_east_1_sh_findings_1_0
    WHERE
        eventday >= '20240115'
        AND eventday <= '20240120'
        AND class_name = 'Authentication'
        AND status_id != 1  -- Failed
),
AggregatedAttempts AS (
    SELECT
        source_ip,
        COUNT(DISTINCT username) AS targeted_users,
        COUNT(*) AS total_attempts,
        ARRAY_AGG(DISTINCT auth_source) AS sources,
        MIN(time) AS first_attempt,
        MAX(time) AS last_attempt
    FROM FailedLogins
    GROUP BY source_ip
)
SELECT
    source_ip,
    targeted_users,
    total_attempts,
    sources,
    first_attempt,
    last_attempt,
    DATE_DIFF('second', first_attempt, last_attempt) AS attack_duration_seconds
FROM AggregatedAttempts
WHERE
    targeted_users > 10
    AND total_attempts > 100
ORDER BY total_attempts DESC;
```

2. Data exfiltration detection:

```
-- Identify large outbound data transfers
WITH BaselineTraffic AS (
    -- Calculate 30-day baseline per source
    SELECT
        src_endpoint.ip AS source_ip,
        AVG(traffic.bytes) AS avg_bytes,
        STDDEV(traffic.bytes) AS stddev_bytes
    FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
    WHERE
        eventday >= '20231220'
        AND eventday < '20240120'
        AND connection_info.direction = 'Outbound'
    GROUP BY src_endpoint.ip
),
CurrentTraffic AS (
    SELECT
        src_endpoint.ip AS source_ip,
        dst_endpoint.ip AS dest_ip,
        SUM(traffic.bytes) AS total_bytes,
        COUNT(*) AS connection_count,
        ARRAY_AGG(DISTINCT dst_endpoint.port) AS dest_ports
    FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
    WHERE
        eventday = '20240120'
        AND connection_info.direction = 'Outbound'
    GROUP BY
        src_endpoint.ip,
        dst_endpoint.ip
)
SELECT
    c.source_ip,
    c.dest_ip,
    c.total_bytes,
    c.total_bytes / 1073741824.0 AS total_gb,
    b.avg_bytes,
    b.stddev_bytes,
    (c.total_bytes - b.avg_bytes) / NULLIF(b.stddev_bytes, 0) AS z_score,
    c.connection_count,
    c.dest_ports
FROM CurrentTraffic c
JOIN BaselineTraffic b ON c.source_ip = b.source_ip
WHERE
    (c.total_bytes - b.avg_bytes) / NULLIF(b.stddev_bytes, 0) > 3  -- >3 std dev
    OR c.total_bytes > 10737418240  -- >10GB absolute threshold
ORDER BY z_score DESC NULLS LAST;
```

3. Lateral movement detection:

```
-- Detect lateral movement patterns
WITH AuthSuccess AS (
    SELECT
        time,
        actor.user.name AS user,
        device.ip AS source_ip,
        dst_endpoint.ip AS target_host
    FROM amazon_security_lake_table_us_east_1_sh_findings_1_0
    WHERE
        eventday = '20240120'
        AND class_name = 'Authentication'
        AND status_id = 1  -- Success
        AND dst_endpoint.ip IS NOT NULL
),
NetworkConnections AS (
    SELECT
        time,
        src_endpoint.ip AS source_ip,
        dst_endpoint.ip AS dest_ip,
        dst_endpoint.port AS dest_port
    FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
    WHERE
        eventday = '20240120'
        AND dst_endpoint.port IN (22, 3389, 445, 5985, 5986)  -- SSH, RDP, SMB, WinRM
)
SELECT
    a.user,
    a.source_ip,
    COUNT(DISTINCT a.target_host) AS unique_targets,
    ARRAY_AGG(DISTINCT a.target_host) AS targets,
    ARRAY_AGG(DISTINCT n.dest_port) AS ports_used,
    MIN(a.time) AS first_lateral,
    MAX(a.time) AS last_lateral
FROM AuthSuccess a
JOIN NetworkConnections n
    ON a.source_ip = n.source_ip
    AND a.target_host = n.dest_ip
    AND n.time BETWEEN a.time AND a.time + INTERVAL '5' MINUTE
```

```
GROUP BY a.user, a.source_ip
HAVING COUNT(DISTINCT a.target_host) > 3  -- Accessed >3 hosts
ORDER BY unique_targets DESC;
```

4. Threat intelligence correlation:

```
-- Match known IOCs across all Security Lake data
WITH ThreatIndicators AS (
    SELECT
        ioc_value,
        ioc_type,
        threat_name,
        severity
    FROM threat_intel_table
    WHERE
        last_seen >= CURRENT_DATE - INTERVAL '30' DAY
        AND active = true
),
AllObservables AS (
    -- Extract observables from all sources
    SELECT
        time,
        eventday,
        class_name,
        observable.value AS ioc_value,
        observable.type_id AS obs_type,
        actor.user.name AS user,
        device.ip AS device_ip,
        metadata.product.name AS source
    FROM amazon_security_lake_table_us_east_1_sh_findings_1_0
    CROSS JOIN UNNEST(observables) AS t(observable)
    WHERE eventday >= '20240115'
)
SELECT
    o.time,
    o.class_name,
    o.source,
    o.user,
    o.device_ip,
    t.ioc_value AS matched_ioc,
    t.ioc_type,
    t.threat_name,
    t.severity
FROM AllObservables o
JOIN ThreatIndicators t ON o.ioc_value = t.ioc_value
ORDER BY t.severity DESC, o.time DESC;
```

Query optimization techniques:

1. Partition pruning (most critical):

```
-- BAD: Full table scan
SELECT * FROM amazon_security_lake_table_us_east_1_cloud_trail_1_0
WHERE time >= '2024-01-15'  -- Scans all partitions!

-- GOOD: Partition-aware
SELECT * FROM amazon_security_lake_table_us_east_1_cloud_trail_1_0
WHERE
    eventday >= '20240115'  -- Partition column
    AND eventday <= '20240120'
    AND time >= '2024-01-15'  -- Additional filter

-- Cost difference:
-- BAD: Scans 1 year = ~365 partitions = ~10 TB scanned
-- GOOD: Scans 6 days = ~6 partitions = ~100 GB scanned
-- Savings: 99% cost reduction
```

2. Columnar optimization (Parquet benefits):

```
-- BAD: SELECT * reads all columns
SELECT * FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
WHERE eventday = '20240120'

-- GOOD: Select only needed columns
SELECT
    time,
    src_endpoint.ip,
    dst_endpoint.ip,
    traffic.bytes
FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
WHERE eventday = '20240120'

-- Cost difference:
-- BAD: Reads 50+ columns
-- GOOD: Reads 4 columns
-- Savings: ~90% data scanned
```

3. Use CTAS for complex queries:

```
-- Create table from expensive query results
CREATE TABLE security_events_summary
WITH (
    format = 'Parquet',
    parquet_compression = 'SNAPPY',
    partitioned_by = ARRAY['event_date'],
    external_location = 's3://analytics-bucket/summary/'
) AS
```

```
SELECT
    DATE(time) AS event_date,
    actor.user.name AS user,
    COUNT(*) AS event_count,
    COUNT(DISTINCT class_name) AS event_types,
    ARRAY_AGG(DISTINCT metadata.product.name) AS sources
FROM amazon_security_lake_table_us_east_1_sh_findings_1_0
WHERE eventday >= '20240101'
GROUP BY DATE(time), actor.user.name;

-- Now query the summary table (much faster and cheaper)
SELECT * FROM security_events_summary
WHERE event_date = DATE '2024-01-20'
AND event_count > 1000;
```

4. Optimize JOIN operations:

```
-- BAD: Large table JOIN large table
SELECT a.*, b.*
FROM large_table_1 a
JOIN large_table_2 b ON a.id = b.id

-- GOOD: Filter then JOIN
WITH FilteredA AS (
    SELECT * FROM large_table_1
    WHERE eventday = '20240120'  -- Reduce before JOIN
),
FilteredB AS (
    SELECT * FROM large_table_2
    WHERE eventday = '20240120'
)
SELECT a.*, b.*
FROM FilteredA a
JOIN FilteredB b ON a.id = b.id;
```

5. Use approximate functions for large datasets:

```
-- GOOD for quick analysis (much faster)
SELECT
    eventday,
    approx_distinct(actor.user.name) AS approx_unique_users,
    approx_percentile(traffic.bytes, 0.95) AS p95_bytes
FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
WHERE eventday >= '20240101'
GROUP BY eventday;

-- vs. EXACT (slower, more expensive)
SELECT
    eventday,
    COUNT(DISTINCT actor.user.name) AS exact_unique_users
FROM amazon_security_lake_table_us_east_1_vpc_flow_1_0
WHERE eventday >= '20240101'
GROUP BY eventday;
```

6. **Query result reuse**:

```
-- Enable query result caching
SET SESSION query_results_s3_access_grants_enabled = true;

-- Identical queries within 24 hours use cached results (no charge)
SELECT COUNT(*) FROM amazon_security_lake_table_us_east_1_cloud_trail_1_0
WHERE eventday = '20240120';
-- First run: Scans data, charges apply
-- Subsequent runs: Uses cache, free
```

7. **Workload management with workgroups**:

```
# Create workgroup for different use cases
aws athena create-work-group \
  --name security-threat-hunting \
  --configuration \
    ResultConfigurationUpdates={
      OutputLocation=s3://athena-results/threat-hunting/
    },\
    EnforceWorkGroupConfiguration=true,\
    PublishCloudWatchMetricsEnabled=true,\
    BytesScannedCutoffPerQuery=1000000000000  # 1TB limit

# Separate workgroups for:
# - Interactive threat hunting (higher limits)
# - Scheduled reports (lower priority)
# - Automated alerting (SLA guarantees)
```

Advanced analytical patterns:

Time-series analysis:

```
-- Detect anomalous patterns over time
WITH HourlyMetrics AS (
    SELECT
        DATE_TRUNC('hour', time) AS hour,
        COUNT(*) AS event_count,
        COUNT(DISTINCT actor.user.name) AS unique_users
    FROM amazon_security_lake_table_us_east_1_cloud_trail_1_0
    WHERE
        eventday >= '20240101'
        AND eventday <= '20240120'
    GROUP BY DATE_TRUNC('hour', time)
),
Statistics AS (
    SELECT
        AVG(event_count) AS avg_count,
        STDDEV(event_count) AS stddev_count
    FROM HourlyMetrics
)
SELECT
    h.hour,
    h.event_count,
    h.unique_users,
    s.avg_count,
    (h.event_count - s.avg_count) / s.stddev_count AS z_score
FROM HourlyMetrics h
CROSS JOIN Statistics s
WHERE ABS((h.event_count - s.avg_count) / s.stddev_count) > 3
ORDER BY z_score DESC;
```

Geospatial analysis:

```
-- Identify impossible travel
WITH RankedLogins AS (
    SELECT
        actor.user.name AS user,
        time,
        device.location.city AS city,
        device.location.coordinates.lat AS lat,
        device.location.coordinates.lon AS lon,
        ROW_NUMBER() OVER (
            PARTITION BY actor.user.name
            ORDER BY time
        ) AS rn
    FROM amazon_security_lake_table_us_east_1_sh_findings_1_0
    WHERE
        eventday = '20240120'
        AND class_name = 'Authentication'
        AND status_id = 1
        AND device.location.coordinates.lat IS NOT NULL
)
SELECT
    curr.user,
    prev.city AS from_city,
    curr.city AS to_city,
    prev.time AS from_time,
    curr.time AS to_time,
    CAST(ACOS(
        SIN(RADIANS(prev.lat)) * SIN(RADIANS(curr.lat)) +
        COS(RADIANS(prev.lat)) * COS(RADIANS(curr.lat)) *
        COS(RADIANS(curr.lon - prev.lon))
    ) * 6371 AS INT) AS distance_km,
    DATE_DIFF('minute', prev.time, curr.time) AS time_diff_minutes,
    CAST(ACOS(
        SIN(RADIANS(prev.lat)) * SIN(RADIANS(curr.lat)) +
        COS(RADIANS(prev.lat)) * COS(RADIANS(curr.lat)) *
        COS(RADIANS(curr.lon - prev.lon))
    ) * 6371 / (DATE_DIFF('minute', prev.time, curr.time) / 60.0) AS INT) AS speed_kmh
FROM RankedLogins curr
JOIN RankedLogins prev
    ON curr.user = prev.user
    AND curr.rn = prev.rn + 1
WHERE DATE_DIFF('minute', prev.time, curr.time) BETWEEN 1 AND 60
HAVING speed_kmh > 800  -- Impossible for commercial flight
ORDER BY speed_kmh DESC;
```

Behavioral profiling:

```
-- Detect users deviating from peer group behavior
WITH UserActivity AS (
    SELECT
        actor.user.name AS user,
        actor.user.department AS department,
        COUNT(*) AS api_calls,
        COUNT(DISTINCT api.operation) AS unique_operations,
        COUNT(DISTINCT cloud.region) AS regions_accessed
    FROM amazon_security_lake_table_us_east_1_cloud_trail_1_0
    WHERE eventday = '20240120'
    GROUP BY actor.user.name, actor.user.department
),
DepartmentBaseline AS (
    SELECT
        department,
        AVG(api_calls) AS avg_calls,
        STDDEV(api_calls) AS stddev_calls,
        AVG(unique_operations) AS avg_ops,
        STDDEV(unique_operations) AS stddev_ops
    FROM UserActivity
    GROUP BY department
)
SELECT
    u.user,
    u.department,
    u.api_calls,
    b.avg_calls AS dept_avg,
    (u.api_calls - b.avg_calls) / NULLIF(b.stddev_calls, 0) AS call_zscore,
    u.unique_operations,
    b.avg_ops AS dept_avg_ops,
    (u.unique_operations - b.avg_ops) / NULLIF(b.stddev_ops, 0) AS ops_zscore
FROM UserActivity u
JOIN DepartmentBaseline b ON u.department = b.department
```

```
WHERE
    ABS((u.api_calls - b.avg_calls) / NULLIF(b.stddev_calls, 0)) > 3
    OR ABS((u.unique_operations - b.avg_ops) / NULLIF(b.stddev_ops, 0)) > 3
ORDER BY call_zscore DESC;
```

Performance monitoring:

```
-- Monitor query performance
SELECT
    query_id,
    query_state,
    state_change_reason,
    submission_date_time,
    completion_date_time,
    engine_execution_time_in_millis,
    data_scanned_in_bytes / 1073741824.0 AS data_scanned_gb,
    total_execution_time_in_millis,
    query_queue_time_in_millis
FROM "information_schema"."queries"
WHERE
    submission_date_time >= CURRENT_DATE - INTERVAL '7' DAY
    AND workgroup = 'security-threat-hunting'
ORDER BY data_scanned_in_bytes DESC
LIMIT 20;
```

Cost optimization best practices:

- Always use partition columns in WHERE clause
- Select only needed columns (avoid SELECT *)
- Use LIMIT for exploratory queries
- Create summary tables for repeated queries
- Use approximate functions for quick analysis
- Enable query result caching
- Set data scanned limits per query
- Monitor and optimize expensive queries
- Consider Athena Federation for external sources
- Use compression (Snappy for Parquet)

Sample cost calculation:

```
Query: 1 TB scanned
Cost: 1000 GB x $5/TB = $5

Optimized query: 10 GB scanned (partition pruning + column selection)
Cost: 10 GB x $5/TB = $0.05
```

```
Savings: 99% ($4.95 per query)
1000 queries/month: $5000 → $50 (saves $4,950/month)
```

Effective Athena usage requires understanding partition pruning, columnar optimization, and analytical patterns - mastering these enables cost-effective, performant security analytics at petabyte scale.

How do you implement real-time alerting and automated response for Security Lake data using Amazon EventBridge and AWS Lambda?

Real-time security automation transforms Security Lake from data repository into active defense platform, enabling immediate detection and response to threats as they occur.

Architecture for real-time alerting:

```
Security Lake (S3) → EventBridge (S3 Event Notifications) → Lambda (Alert Logic)
                                   □
                     SNS/SQS/Step Functions
                                   □
                    Response Actions (Block IP, Disable User, etc.)
```

EventBridge integration patterns:

Pattern 1: S3 event-driven alerting:

```
// EventBridge rule for new Security Lake data
{
  "source": ["aws.s3"],
  "detail-type": ["Object Created"],
  "detail": {
    "bucket": {
      "name": ["aws-security-data-lake-us-east-1-123456789012"]
    },
    "object": {
      "key": [{
        "prefix": "ext/aws_cloudtrail/"
      }]
    }
  }
}
```

Lambda function for real-time CloudTrail analysis:

```python
import json
import boto3
import gzip
from io import BytesIO

s3 = boto3.client('s3')
sns = boto3.client('sns')
dynamodb = boto3.resource('dynamodb')

ALERT_TOPIC_ARN = "arn:aws:sns:us-east-1:123456789012:security-alerts"
HIGH_RISK_ACTIONS = [
    'DeleteBucket',
    'PutBucketPolicy',
    'DeleteDBInstance',
    'DeleteTrail',
    'StopLogging',
    'DeleteFlowLogs',
    'DisableSecurityHub',
    'DeleteDetector'  # GuardDuty
]

def lambda_handler(event, context):
    """
    Real-time analysis of CloudTrail events in Security Lake
    Triggered by S3 object creation
    """
    # Get S3 object details
    bucket = event['detail']['bucket']['name']
    key = event['detail']['object']['key']

    # Download and parse Parquet file
    events = read_parquet_from_s3(bucket, key)

    # Analyze events
    alerts = []
    for event_data in events:
        # Check for high-risk actions
        if is_high_risk_action(event_data):
            alert = create_alert(event_data, 'HIGH_RISK_ACTION')
            alerts.append(alert)

        # Check for unusual access patterns
        if is_unusual_access(event_data):
            alert = create_alert(event_data, 'UNUSUAL_ACCESS')
            alerts.append(alert)

        # Check for credential exposure
        if contains_exposed_credentials(event_data):
            alert = create_alert(event_data, 'CREDENTIAL_EXPOSURE')
            alerts.append(alert)
            # Immediate response
            rotate_credentials(event_data)

    # Send alerts
    if alerts:
        send_alerts(alerts)
        store_alerts(alerts)

    return {
        'statusCode': 200,
        'alerts_generated': len(alerts)
    }

def is_high_risk_action(event):
    """Detect high-risk AWS API calls"""
    api_operation = event.get('api', {}).get('operation')

    if api_operation in HIGH_RISK_ACTIONS:
        # Additional context checks
        actor = event.get('actor', {}).get('user', {}).get('name')

        # Alert if action by non-admin or from unusual IP
        if not is_authorized_admin(actor):
            return True

        source_ip = event.get('device', {}).get('ip')
        if not is_known_ip(source_ip):
            return True

    return False

def is_unusual_access(event):
    """Detect anomalous access patterns using DynamoDB baseline"""
    user = event.get('actor', {}).get('user', {}).get('name')
    region = event.get('cloud', {}).get('region')
    time_hour = int(event.get('time', '').split('T')[1].split(':')[0])

    # Get user's normal behavior from DynamoDB
    table = dynamodb.Table('user-behavior-baseline')
    response = table.get_item(Key={'user': user})

    if 'Item' not in response:
        return False  # New user, no baseline

    baseline = response['Item']

    # Check region
    if region not in baseline.get('normal_regions', []):
        return True
    # Check time of day
    if time_hour not in baseline.get('normal_hours', []):
        return True

    return False

def contains_exposed_credentials(event):
    """Detect accidental credential exposure"""
    # Check for GetSecretValue calls
    if event.get('api', {}).get('operation') == 'GetSecretValue':
        # Check if logged without sanitization
        request_params = event.get('unmapped', {}).get('requestParameters', {})

        if 'secretString' in str(request_params):
            return True

    # Check for credential-related errors
    error_message = event.get('unmapped', {}).get('errorMessage', '')
    if any(keyword in error_message.lower() for keyword in ['password', 'secret',
'key', 'token']):
        return True

    return False

def create_alert(event, alert_type):
    """Create structured alert from event"""
    return {
        'alert_id': f"{alert_type}-{event.get('metadata', {}).get('uid')}",
        'alert_type': alert_type,
        'severity': 'HIGH' if alert_type in ['CREDENTIAL_EXPOSURE',
'HIGH_RISK_ACTION'] else 'MEDIUM',
        'timestamp': event.get('time'),
        'user': event.get('actor', {}).get('user', {}).get('name'),
        'source_ip': event.get('device', {}).get('ip'),
        'action': event.get('api', {}).get('operation'),
        'resource': event.get('resources', [{}])[0].get('uid') if
event.get('resources') else None,
        'region': event.get('cloud', {}).get('region'),
        'account': event.get('cloud', {}).get('account', {}).get('uid'),
        'raw_event': json.dumps(event)
    }

def send_alerts(alerts):
    """Send alerts via SNS"""
    for alert in alerts:
        message = format_alert_message(alert)

        sns.publish(
            TopicArn=ALERT_TOPIC_ARN,
            Subject=f"Security Alert: {alert['alert_type']}",
            Message=json.dumps(message, indent=2),
            MessageAttributes={
                'severity': {
                    'DataType': 'String',
                    'StringValue': alert['severity']
                },
                'alert_type': {
                    'DataType': 'String',
                    'StringValue': alert['alert_type']
                }
            }
        )

def store_alerts(alerts):
    """Store alerts in DynamoDB for tracking"""
    table = dynamodb.Table('security-alerts')

    with table.batch_writer() as batch:
        for alert in alerts:
            batch.put_item(Item={
                **alert,
                'ttl': int(time.time()) + (90 * 86400)  # 90-day retention
            })
```

Pattern 2: Scheduled analysis with EventBridge cron:

```
# EventBridge rule: Run every 5 minutes
# Schedule expression: rate(5 minutes)

def scheduled_threat_detection(event, context):
    """
    Periodic analysis of recent Security Lake data
    Detects patterns requiring correlation over time
    """
    athena = boto3.client('athena')

    # Query for brute force attempts (last 5 minutes)
    query = """
    SELECT
        actor.user.name AS user,
        device.ip AS source_ip,
        COUNT(*) AS failed_attempts,
        ARRAY_AGG(DISTINCT metadata.product.name) AS sources
    FROM amazon_security_lake_table_us_east_1_sh_findings_1_0
    WHERE
        eventday = CAST(CURRENT_DATE AS VARCHAR)
        AND time >= CURRENT_TIMESTAMP - INTERVAL '5' MINUTE
        AND class_name = 'Authentication'
        AND status_id != 1
    GROUP BY actor.user.name, device.ip
    HAVING COUNT(*) > 20
    """

    results = execute_athena_query(query)

    # Process results
    for row in results:
        alert = {
            'alert_type': 'BRUTE_FORCE',
            'severity': 'HIGH',
            'user': row['user'],
            'source_ip': row['source_ip'],
            'failed_attempts': row['failed_attempts'],
            'sources': row['sources']
        }

        # Automated response
        block_ip(row['source_ip'])
        disable_user(row['user'])

        send_alert(alert)
```

Pattern 3: Multi-stage attack detection:

```
import boto3
from datetime import datetime, timedelta

stepfunctions = boto3.client('stepfunctions')

def detect_attack_chain(event, context):
    """
    Correlate events detecting multi-stage attacks
    Uses Step Functions for stateful correlation
    """
    # Parse incoming event
    security_event = parse_security_lake_event(event)

    # Start or update attack correlation workflow
    execution_arn = start_correlation_workflow(security_event)

    return {'execution_arn': execution_arn}

def start_correlation_workflow(event):
    """
    Step Functions workflow correlating events over time
    """
    state_machine_arn = "arn:aws:states:us-east-
1:123456789012:stateMachine:AttackCorrelation"

    response = stepfunctions.start_execution(
        stateMachineArn=state_machine_arn,
        input=json.dumps({
            'event': event,
            'correlation_window': 3600,  # 1 hour
            'stages': {
                'reconnaissance': None,
                'initial_access': None,
                'execution': None,
                'persistence': None,
                'privilege_escalation': None,
                'defense_evasion': None,
                'credential_access': None,
                'discovery': None,
                'lateral_movement': None,
                'collection': None,
                'exfiltration': None
            }
        })
    )

    return response['executionArn']
```

Step Functions state machine (attack correlation):

```
{
  "Comment": "Multi-stage attack correlation",
  "StartAt": "ClassifyEvent",
  "States": {
    "ClassifyEvent": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-
1:123456789012:function:ClassifyMITRETactic",
      "Next": "UpdateAttackChain"
    },
    "UpdateAttackChain": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-
1:123456789012:function:UpdateAttackTimeline",
      "Next": "CheckAttackProgress"
    },
    "CheckAttackProgress": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.attack_stage_count",
          "NumericGreaterThanEquals": 3,
          "Next": "AlertHighConfidenceAttack"
        }
      ],
      "Default": "WaitForMoreEvents"
    },
    "AlertHighConfidenceAttack": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:SendCriticalAlert",
      "End": true
    },
    "WaitForMoreEvents": {
      "Type": "Wait",
      "Seconds": 300,
      "Next": "QueryRecentEvents"
    },
    "QueryRecentEvents": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:QuerySecurityLake",
      "Next": "CheckAttackProgress"
    }
  }
}
```

Automated response actions:

1. IP blocking:

```
def block_ip_automated(source_ip, reason):
    """Block malicious IP in AWS WAF and Security Groups"""
    waf = boto3.client('wafv2')
    ec2 = boto3.client('ec2')

    # Add to WAF IP set
    waf.update_ip_set(
        Name='BlockedIPs',
        Scope='REGIONAL',
        Id='ipset-id',
        Addresses=[f"{source_ip}/32"],
        LockToken='token'
    )

    # Update security groups
    security_groups = ec2.describe_security_groups(
        Filters=[{'Name': 'tag:AutoBlock', 'Values': ['true']}]
    )

    for sg in security_groups['SecurityGroups']:
        ec2.revoke_security_group_ingress(
            GroupId=sg['GroupId'],
            IpPermissions=[{
                'IpProtocol': '-1',
                'IpRanges': [{'CidrIp': f"{source_ip}/32"}]
            }]
        )
    # Log action
    log_response_action({
        'action': 'BLOCK_IP',
        'ip': source_ip,
        'reason': reason,
        'timestamp': datetime.utcnow().isoformat()
    })
```

2. User account suspension:

```
def disable_compromised_user(username, reason):
    """Disable user account across AWS and federated systems"""
    iam = boto3.client('iam')

    # Disable IAM user
    try:
        # Delete access keys
        keys = iam.list_access_keys(UserName=username)
        for key in keys['AccessKeyMetadata']:
            iam.delete_access_key(
                UserName=username,
                AccessKeyId=key['AccessKeyId']
            )

        # Attach deny-all policy
        iam.attach_user_policy(
            UserName=username,
            PolicyArn='arn:aws:iam::aws:policy/AWSDenyAll'
        )

        # Revoke active sessions
        iam.delete_login_profile(UserName=username)

    except iam.exceptions.NoSuchEntityException:
        pass  # Not an IAM user

    # Disable in Okta (if federated)
    disable_okta_user(username)

    # Create incident ticket
    create_incident(
        title=f"User Disabled: {username}",
        description=f"Automated response: {reason}",
        severity='HIGH'
    )
```

3. Resource isolation:

```
def isolate_compromised_instance(instance_id, reason):
    """Isolate EC2 instance for forensics"""
    ec2 = boto3.client('ec2')

    # Create forensic security group (deny all)
    forensic_sg = ec2.create_security_group(
        GroupName=f'forensic-{instance_id}',
        Description='Isolated for forensic analysis',
        VpcId='vpc-xxxxx'
    )

    # Replace instance security groups
    ec2.modify_instance_attribute(
        InstanceId=instance_id,
        Groups=[forensic_sg['GroupId']]
    )

    # Create snapshot for forensics
    volumes = ec2.describe_volumes(
        Filters=[{'Name': 'attachment.instance-id', 'Values': [instance_id]}]
    )

    for volume in volumes['Volumes']:
        ec2.create_snapshot(
            VolumeId=volume['VolumeId'],
            Description=f'Forensic snapshot - {reason}',
            TagSpecifications=[{
                'ResourceType': 'snapshot',
                'Tags': [
                    {'Key': 'Forensic', 'Value': 'true'},
                    {'Key': 'InstanceId', 'Value': instance_id},
                    {'Key': 'Reason', 'Value': reason}
                ]
            }]
        )

    # Tag instance
    ec2.create_tags(
        Resources=[instance_id],
        Tags=[
            {'Key': 'Status', 'Value': 'Isolated'},
            {'Key': 'IsolationReason', 'Value': reason}
        ]
    )
```

Alerting integrations:

SNS to Slack:

```
import json
import urllib3

http = urllib3.PoolManager()

def lambda_handler(event, context):
    """
    Forward SNS alerts to Slack
    """
    message = json.loads(event['Records'][0]['Sns']['Message'])
    severity = event['Records'][0]['Sns']['MessageAttributes']['severity']['Value']

    # Color-code by severity
    color = {
        'CRITICAL': '#FF0000',
        'HIGH': '#FFA500',
        'MEDIUM': '#FFFF00',
        'LOW': '#00FF00'
    }.get(severity, '#808080')

    slack_message = {
        "attachments": [{
            "color": color,
            "title": f"□ Security Alert: {message['alert_type']}",
            "fields": [
                {
                    "title": "Severity",
                    "value": severity,
                    "short": True
                },
                {
                    "title": "User",
                    "value": message.get('user', 'Unknown'),
                    "short": True
                },
                {
                    "title": "Source IP",
                    "value": message.get('source_ip', 'Unknown'),
                    "short": True
                },
                {
                    "title": "Action",
                    "value": message.get('action', 'Unknown'),
                    "short": True
                }
            ],
            "footer": "AWS Security Lake",
            "ts": int(datetime.now().timestamp())
        }]
    }
    http.request(
        'POST',
        os.environ['SLACK_WEBHOOK_URL'],
        body=json.dumps(slack_message),
        headers={'Content-Type': 'application/json'}
    )
```

Best practices:

- Implement tiered alerting (CRITICAL → PagerDuty, MEDIUM → Slack)
- Use Step Functions for stateful correlation
- Store alert history in DynamoDB with TTL
- Implement alert deduplication
- Test automated responses in non-production first
- Maintain manual override capability
- Comprehensive logging of all automated actions
- Regular review of false positives
- Tune detection thresholds based on baseline
- Document runbooks for each alert type

Real-time automation transforms Security Lake from passive repository into active defense system, enabling immediate threat detection and response at cloud scale with minimal human intervention.