FIRST Big Data Framework for Incident Detection and Response

Version 1

Editors: Steve McKinney (Cisco) and Ben Rhodes (Cisco)
Contributors: Erik Ivan Cruz (Totalsec), Jamison Day (LookingGlass), Michael Graber (Open Systems AG), Steve McKinney (Cisco), Ben Rhodes (Cisco), Morton Swimmer (Trend Micro), Gavin Reid (Recorded Future), Allan Thomson (LookingGlass)

The Goal of this Document

The amount of security-relevant data for the organizations we protect is growing beyond the capabilities of traditional incident detection and response tools. Evaluating and operationalizing the “big data” technologies capable of storing and analyzing data at scale requires technical depth uncommon to IR (Incident Response) teams and is non-trivial and time consuming. Security teams, such as IR, vulnerability management, architecture, etc, need both the ability to deploy and store data in these technologies, and to use them to enable “playbooks” for detection and response.

This document will cover the concepts and technologies used in the big data space and detail how they have been successfully leveraged to scale IDR (Incident Detection and Response) environments. It will also cover technologies this working group has tried and found not to scale or meet IDR needs.

The goal of this document is to help IDR teams determine if big data technologies make sense for their environment and if so, the jump start them and eliminate much of the toil the working group has learned from.

Why is Big Data Relevant to IDR?

The term big data is often misused or used to represent many different concepts depending on who is delivering the message or what context that data is being used for. In its simplest definition, big data refers to data sets that are too large for traditional data processing techniques to manage or manipulate. For some organizations, it may be hundreds of gigabytes and others it may be hundreds of terabytes that require a change in how data is managed. Within cyber security, the term is often used to represent a data set that can influence or provide improved security outcomes to organizations where they are able to leverage the data set in ways that were not previously possible.

A key aspect of understanding the concept of big data within the cyber security world is how the data is collected, stored, updated, indexed, searched, shared, analyzed and visualized. Therefore to help define what big data is, we start by defining some use cases where larger data sets can enhance the security of an organization.

Situational Awareness

One of the most urgent and operational requirements for security teams, whether they are building an understanding of activities of threat actors or actively responding to incidents, is building situational awareness. The Internet is a vast universe of connected devices, applications and users. Big data technologies can enable organizations to collect an awareness of that universe. This can involve billions or trillions of data points and represent the connections between them as well as relevant security metadata.

Historical Context

The cybersecurity world is a continuously changing environment. The Internet, devices, networks, applications and users connecting to that world are in a continuous state of change. Historical context can enable security team to understand how that world changes over time and potentially model those changes to help shape controls and response. Big data enables organizations to represent and handle the events and associated times that occur in that universe. This helps answer questions based on a timeframe of when events occurred, when data was discovered, how urgently specific data must be treated and so on.

Operational Use

Big data technologies can help IDR teams efficiently search over a large corpus of security data. This capability is critical in determining whether an event or data fragment is part of a larger threat or is benign. Such searches are common and often must complete in seconds or minutes, not days. The search problem is non-trivial as it depends on what you are searching for and how you are searching. For instance, efficient search over IP addresses requires a different approach than efficient search over unstructured text. But, it is possible to address these issues with the right technologies and data architecture.

Retrospective Detection

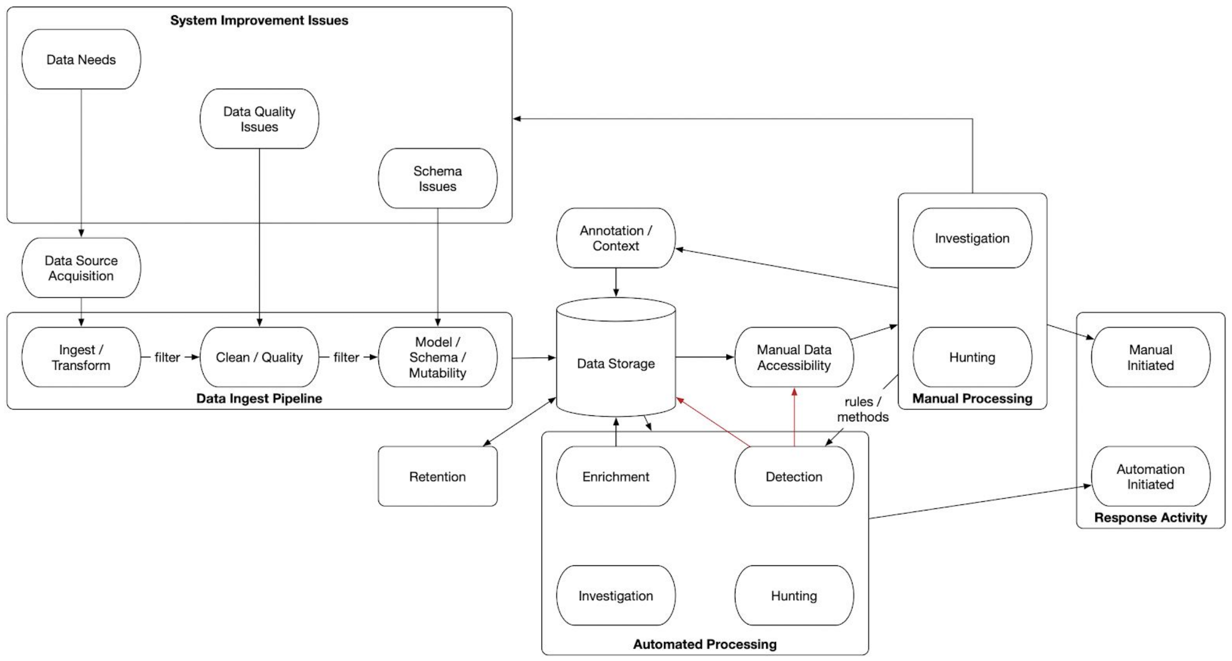
New threat indicators are found every day, often for attacks which occurred days or months in the past. Big data allows organizations to store logs for longer periods of time and periodically re-analyze this historical data to identify previously unknown events in your network.

Data Enrichment

Investigations often start from a single event which triggered an alert. These events rarely have sufficient context to determine the source of a threat, the scope of the incident, and business impact. Big data technologies make it possible to enrich these events with other data sources prior to a responder picking up the case which gives context and eliminates what is often human effort in gathering needed information.

Architecture Summary

A big data system architecture may be summarized as shown below:



There are six main areas of the architecture that are important to define and understand.

1) Data Ingest Pipeline
    a) The set of processing that takes raw data collected and through a sequence and connected set of steps turns the data into a format that may be stored in a large data store
    b) At the same time, controls must be in place to detect data drift in form of changing schemata, changing value ranges or semantics.
2) Data Storage and Accessibility
    a) The database and storage technologies used to persist data from the ingest pipeline that are accessible to users for automated and manual processing

3) Manual Processing
    a) The processing of data by humans where machine’s have not yet been coded (due to cost, complexity or other challenges) to automate correlations or insight that human beings possess within their intelligence field.
4) Automated Processing
    a) The set of steps and processes that continuously evaluate, refine and enrich data towards outcomes that either a) require further manual processing by a human being or b) determined action that feeds into a response or update to security
    b) Automated processing may include modules that span from simple processing such as data matching and enrichment; to more complex processing that leverages machine learning techniques to find data connections; patterns and anomalies that otherwise are not easily found by simple processing techniques.
5) Response Activity
    a) The processing or actions taken by human or automated outcomes
6) System Improvements
    a) The set of requirements and challenges on how data is collected; data is refined and modelled for the set of use cases intended by the big data system

Data Ingest Pipeline

Once data sources have been identified, the data needs to be ingested into the platform. Data ingestion encompasses the pipeline stages required to make data readily available for manual and automated processing. This can appear minimally important at first glance, but it is a critical foundation to enable more meaningful and efficient usage of the data. Specifically, this includes initial ingest, transformation, quality checks, filtering, schema validation, and persistence to a data storage technology. The pipeline components typically must be able to handle many technologies and data formats, as data sources come in many forms.

We will explore how to get started in this area, covering core concepts and common pitfalls, focused on considerations to make in a batch implementation. A streaming approach can also be applied to enable lower latency, horizontal scalability, and a microservice approach. But, streaming is a more advanced topic that will not be covered in this document.

Establish Standards

Establishing a baseline of standards can set an easier path for implementation and long-term maintenance. Shared components can be built up to automate these standards and enable re-use. And as standards evolve, you can update those components and apply them to all pipelines. Be very intentional here and minimize choices, to enable a manageable ecosystem. Here's an example to get you started:

1) Data Technology

All data will be stored in HDFS, accessible by Spark and Hive, with the exception of datasets that are best represented in binary form.

2) Directory Structure Pattern

Note, that there are a few considerations to make here, but the point is to select a standard that works for your organization. For example, year=xxxx/month=xx/day=xx may be necessary for some file systems due to performance. If using Spark, however, a flatter hierarchy is necessary to uncover the full power of Spark partitioning. In that case, the trade-off is worth it if the underlying filesystem doesn't have issues with large directory listings. Below is one example of a standard directory structure:

```
/data/
  <category>/
    <organization>/
      <dataset>/
        <format_version>/
          ingest_date=<YYYY-MM-DD>/
            ingest_hourstamp=<utc_epoch_time>
```

This path must only contain lowercase alphanumeric characters and dashes.

`<category>` must be one of the following:
[dns, enrichment, reputation, vulnerabilities, ...]

`<organization>` is optional and when used to indicate the org that owns the data

`<dataset>` is the name of the dataset

`<format>` must be one of the following:
[json, parquet, binary, ...]

`<version>` indicates version of the data schema
Example: v1

`<YYYY-MM-DD>` - year, month and day
Example: 2019-06-10

`<utc_epoch_time>` is optional
Can be useful for large datasets that need additional partitioning (>100GB/day).

Note, that 'ingest_hourstamp' is favored over 'hourstamp' to allow for hourstamp attribute within the data.

Example:
`/data/enrichment/alexa/csv_v2/ingest_date=2019-06-12`

3) Data Owner and Classification

Each dataset will have an assigned data owner that is accountable for selecting data classification and approving access requests.

4) Field Naming (Normalization)

The following commonly used fields should be named and formatted consistently across all datasets, when used:

- `source_ip_addr` - source IP address
- `dest_ip_addr` - destination IP address
- `first_seen` - time that the record content was first observed, in UTC epoch time
- `last_seen` - time that the record content was last observed, in UTC epoch time
- ...

5) Schema Validation

Before data is made readily available for users, it should be validated against a known schema that is defined by a JSON Schema definition or a Python Data Class. This is optional, since it is difficult to achieve for some datasets as well as when using a batch approach. The depth of validation is left to the discretion of the engineer, given the complexity of the particular dataset and performance impact. When using JSON Schema validation, the `fastjsonschema` Python package should be used for best performance.

6) Retention

Each dataset should have retention rules defined and automatically enforced.

Ingestion and Transformation

Ingestion is fetching data from a remote source into the platform. This can be as simple as periodically pulling a file from a web server or more complex such as streaming data from a message queue. Sometimes data needs to be transformed into a different format to simplify data pipelines or access to the data. Common examples are uncompressing a zipped file and transforming a CSV file to JSON.

Initial Ingest of Raw Data

In a batch system, it is recommended to establish an initial raw ingest stage that persists the raw data into the target file system, but in a protected area. As mentioned in the section above, many of the established standards around ingest can be implemented in components that are reused across pipelines. If using Python, this can come in the form of developing a core Python package that can be imported and used for specific pipelines. The same concepts apply regardless of the language used however. Also, as you implement components that will be shared, select a package versioning approach (recommended: https://semver.org/).

Components to Consider:

- FileSystem - Encapsulate functionality around target file system (ex. get_target_path)
- Rsync/SSH - rsync to target file system
- RESTful API's - API consumption to target file system
- NFS - move/copy files in NFS dir to target file system
- S3 - move/copy files in S3 to file system
- Database Query - create file in target file system, based on query results
- MetricsAndLogging - standardize what/how/where logs and metrics are captured

Transforming Data for Use

Once data has been onboarded, a separate batch transform stage should be applied to a) align data to established standards, b) filter out unnecessary data and c) perform any desired aggregations. Select a technology that can be consistently used for transforms, such as Apache Spark. Spark provides a wealth of analytic processing features that go beyond the basic transform needs, and such is recommended if you are using an ecosystem that supports Spark (Hadoop, Mesos, or Kubernetes, accessing data in HDFS, Cassandra, Hive, etc, using languages such as Python, Java, Scala, and R).

When transforming data, it's important to also consider how to track what's been transformed already. Transform failures can occur, as well as be intermittent, caused by data issues, platform issues, etc. Although monitoring/alerting could be created around all failures, it can reduce operational burden to have a component that keeps track of successful transforms (per dataset, transform, and time range). With such a component, transforms can automatically be run based on time ranges that have not been successfully performed yet.

Cleaning and Quality Validation

Occasionally the data ingested will be malformed or contain data quality issues such as missing values. These need to be detected, and potentially corrected, in the pipeline. Common issues, along with examples, include:

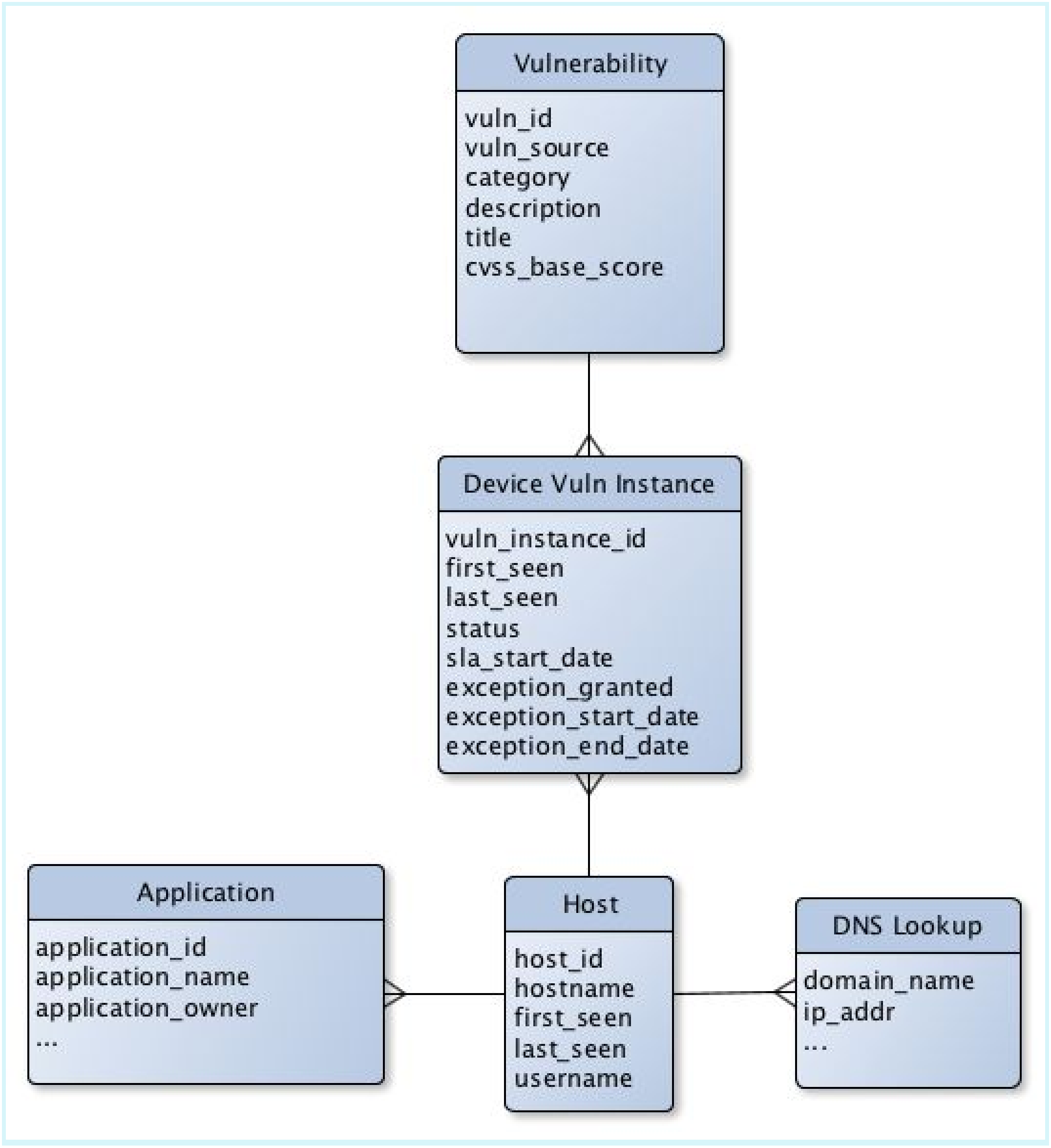
- Malformed data – JSON was expected but a syntax error prevented parsing of the data
- Missing fields – A username field should have been in a record, but was not present
- Missing or Null values – The field for an IP address is present and expected, but there is no IP address in the record
- Impossible values – A record has a timestamp from the future
- Data drift - In some of the aforementioned cases, the reason is because the producer of the data has changed something. As this can invalidate any decision made using the data, it must be flagged and examined.

Often, issues during the data validation stage can lead to discovery of issues occurring at the source of the data. For instance, impossible timestamps may indicate NTP is not synchronized in the data collection infrastructure.

Data cleaning also includes normalization of fields and values. Different data sets may have differing labels for the same logical field. For instance, a timestamp in one data set may be called `timestamp`, while in another is called `ts`. Normalization addresses this issue by renaming the fields to a common label which simplifies downstream work like joining across datasets. It can also be useful in standardizing data values, such as ensuring all MAC addresses are upper case.

Data Modeling and Schema Management

Data modeling is a process that can help ensure the data being collected can be used to solve problems within an organization. It structures the mapping of fields and values within individual data sets as well as relationships across data sets which can be useful for connecting them. For instance, the model below shows the data model relationships among vulnerabilities, devices, and applications.



Schema management enables changes in the structure of data sets to be tracked over time. Software updates or technology changes can result in these structure changes which can break data ingestion pipelines. Well-defined schemas enable users of the data to understand what a data set contains, if it is useful for their problem, and how to interact with the data. Below is a partial schema from a VPN data source written in JSON Schema [json-schema.org]:

```
{
    "$schema": "http://json-schema.org/draft-04/schema",
    "type": "object",
    "properties": {
        "timestamp": {"type": "string", "description": "Timestamp of record"},
        "username": {"type": "string", "description": "Username associated with the IP
address"},
        "ip": {"type": "string", "description": "IPv4 address assigned"},
        "ipv6": {"type": "string", "description": "IPv6 address assigned "},
        ...
    },
    "anyOf": [
        {"required": ["username", "timestamp", "ip"]},
        {"required": ["username", "timestamp", "ipv6"]},
        ...
    ],
    ...
}
```

This schema defines property names (e.g. `username`), types (e.g. `number`, `string`), descriptions, and information about well-formed records (e.g. a username, timestamp, and either an IPv4 or IPv6 address are required for a record to conform to the schema).

Beyond schemas, there are also taxonomies and ontologies of data that have been a recent interest. These attempt to define the semantics of these fields and avoid misunderstandings and enable reasoning. However, this topic is outside the scope of this version of the paper.

Monitoring/Logging

In order to reliably monitor the data pipelines, select an appropriate tool for collecting logs, events, and metrics. Listed below are a few examples of tools in this realm.

- Prometheus (open source) - collects metrics and has alerting support
- Nagios (open source) - log management and alerting
- DataDog - collects metrics and events, with alerting support
- Splunk - collects logs (and effectively events) and metrics, with alerting support
- PagerDuty - streamlines workflow around alerts

Types of monitors to consider:

- Latency - alerts when data hasn't been seen in X time
- Errors - excessive / non-recoverable errors during ingest or transform
    - Non-data Errors - errors regarding the movement of data
    - Record Validation Errors - errors regarding individual record validation
- Saturation - excessive backlog of ingest or transform jobs that is growing

If the types of metrics and events collected are consistent, then monitors can also be consistently applied across all pipelines.

Retention

If the directory standards are followed above, a retention component can relatively easily be created and used across all pipelines. Configuration items to consider per pipeline:

- mode
    - delete - delete data found outside of retention window
    - trash - move data into hidden trash directory to be auto deleted
    - archive - move data into archived directory that can still be explored by users, but not with any performance expectations
- days to retain - number of days to keep data in primary storage location
- base directory - base directory of dataset
- archive directory - optional base directory to use for placement of archived data

Many countries consider logfiles to be a type of personal data and require organisations to define, declare and implement a retention period for them: either as a fixed time after they are collected, or after some other event (e.g. employee leaves). At the end of this period the logs must be deleted, aggregated or anonymised. Typically, the organisation can set its own retention period (though “for ever” is unlikely to be acceptable), so long as the choice can be justified.

A good way to both determine and justify the appropriate retention period is to consider your reasons and processes for using logs: how long after the event would these be meaningful? At some point in time logs will lose their value, for example because of infrastructure or software changes – whether inside or outside the organisation – or because an intruder will have had ample time to do all the damage they wish. Guidelines from regulators and industry groups may provide a useful comparison: if your conclusions are very different to theirs, work out why.

Note that keeping logs for too long may not only be a legal problem, but a reputational and practical one. Several recent news stories have covered breaches with consequences going back more than a decade, apparently because the organisations concerned either did not have, or did not implement, a retention policy.

Data Storage and Accessibility

At the center of the architecture diagram is data storage. This is where the data ingest pipeline persists data for manual and automated data usage. In the simplest architecture, this may be a single technology, but there can often be a need for multiple depending on the use cases surrounding the data.

Storage

There are a number of database/storage types to consider and there is no single combination today that can be easily prescribed. So, when selecting which storage technology to use, consider cost, operational burden, performance, stability, and compatibility with other technologies. And perhaps most importantly, consider ease of use with the known use cases in mind.

Here are the most common database/storage types available:

- Distributed File System (ex. Hadoop File System)

- Object Storage (ex. S3)
- Key/Value Database
    - Riak is scalable, stable and has a time-series optimised variant.
- Column Oriented Database
    - HBase
    - Cassandra
    - AWS DynamoDB
- Document Database
    - ElasticSearch
- Graph Database
    - There are many graph databases with different properties, each should be tested to ensure they scale for your use cases (StarDog, Neo4J, Ontotext, ArangoDB).
- Relational Database
    - CockroachDB
    - PostgreSQL, which can scale for some types of applications
- Analytical Database
    - Clickhouse is a columnar database that has been tested to trillions of records and millions of inserts per second. It is effective for storing telemetry and time series data. It supports a variant of SQL and scales well linearly and horizontally.
- In-Memory Database
    - Redis
    - Memcached

In big data applications, column stores like Cassandra have proven themselves as being scalable and resilient. They usually allow a key-value like object to be stored with schema flexibility and the column orientation allows the data to be sharded and duplicated to adapt to different read/write performance requirements. Document databases have their uses for more complex objects, but there are scalability/performance tradeoffs. Elasticsearch is slow with writing, while CouchDB is fast at reading and writing, but doesn't scale. Graph databases are often a great way of representing knowledge, but scaling them can be tricky. Relational databases have a bad, but often unjustified, reputation for not scaling well. Often it is not the RDMS at fault but the data model that was poorly designed.

Elasticsearch and Solr can also be used as indices to other data to speed up retrieval or allow for keyword searches in unstructured data.

Memory data stores, like Memcached or Redis also have their place in big data architecture. They are often used for caching, de-duplicating data, distributed interprocess coordination and near-line data storage amongst other things. They should not be used as permanent data storage.

Accessibility

We have found that using Jupyter notebooks for performing computation, documenting and recording results of analyses helps in spreading knowledge. Jupyter is not restricted to just Python any more and supports a variety of languages. Sharing can happen in multiple ways. The notebook file can be sent around or stored on a shared file system for others to use. Jupyter can also be set up to allow access to multiple users, though this is not advisable. Better is to use JupyterHub which is a proper multi-tenant solution and allows more sensible access control. Some cloud platforms also offer ways of sharing code. Zeppelin offers a similar notebook capability.

Manual Processing

Manual processing in a big data environment often requires different sets of tools to enable analysis at scale. Processing frameworks such as Map Reduce or Spark are common and often the starting point. Spark has proven to be flexible and sufficient for working with data in HDFS and is the tool used in this section.

Spark SQL is easy to learn given a background in a SQL dialect and it allows interaction with the data as if it were a table in a database. Spark is optimized for Scala, but many users prefer to use Python with Spark (PySpark) due to its simplicity. Below are several examples leveraging Spark with passive DNS query logs.

```
# Start a PySpark Shell
% pyspark

# Load the passive DNS data from JSON files
>>> pdns_json = spark.read.json('/hdfs/path/to/data/')

# Create a SQL table from the data
pdns_json.registerTempTable('pdns')

# See how many records there are
>>> spark.sql('SELECT COUNT(*) FROM pdns').show()
+--------+
|count(1)|
+--------+
|68080509|
+--------+
# View the schema of the table
>>> spark.sql('DESCRIBE TABLE pdns').show()
+---------+---------+-------+
| col_name|data_type|comment|
+---------+---------+-------+
|client_ip|   string|   null|
|   qclass|   string|   null|
|    qname|   string|   null|
|    qtype|   string|   null|
|timestamp|   bigint|   null|
+---------+---------+-------+

# Count the number of client IP addresses in the network which
# have queried for `www.first.org.`
>>> spark.sql('SELECT COUNT(DISTINCT(client_ip)) FROM pdns WHERE
qname="www.first.org."').show()
+-------------------------+
|count(DISTINCT client_ip)|
+-------------------------+
|                        3|
+-------------------------+

# Show the occurrences of www.first.org. being queried.  This is
# useful for retrospective detections.
>>> spark.sql('SELECT * FROM pdns WHERE
qname="www.first.org."').show()
+----------+------+--------------+-----+----------+
| client_ip|qclass|         qname|qtype| timestamp|
+----------+------+--------------+-----+----------+
|   1.2.3.4|    IN|www.first.org.|    A|1557356910|
|   1.2.3.4|    IN|www.first.org.|    A|1557360241|
|   1.2.5.6|    IN|www.first.org.|    A|1557356630|
|   1.2.3.9|    IN|www.first.org.|    A|1557360213|
+----------+------+--------------+-----+----------+

# Show the lookups made by a given client.  This is useful when
# investigating what happened prior to a detection.
>>> spark.sql('SELECT timestamp, qname FROM pdns WHERE
client_ip="1.2.3.45" AND timestamp >= 1557356900 ORDER BY
timestamp').show(5)
+-----------+-------------------------+
|  timestamp|                    qname|
+-----------+-------------------------+
| 1557356910|              google.com.|
| 1557356910|          www.google.com.|
| 1557356912|      a6281279.yolox.net.|
| 1557356912| ww17.a6281279.yolox.net.|
| 1557356913|  ww1.a6281279.yolox.net.|
+-----------+------------------------+

# Find the top queried names.  This can be used to build a custom
list
# of qname popularity for an organization which can serve as
# context during investigations.
>>> spark.sql('SELECT qname, COUNT(*) AS count FROM pdns GROUP BY
qname ORDER BY count DESC').show(5)
+---------------+-------+
|qname          |count  |
+---------------+-------+
|pool.ntp.org.  |3214642|
|www.google.com.|3143562|
|microsoft.com. |3136000|
|apple.com.     |2853032|
|amazonaws.com. |2081906|
+---------------+-------+
```

References for learning more

- Installing Spark - https://spark.apache.org/downloads.html
- Getting Started with Spark - https://spark.apache.org/docs/latest/
- Getting Started with Spark SQL -
  https://spark.apache.org/docs/latest/sql-getting-started.html
- What is MapReduce? - https://www.guru99.com/introduction-to-mapreduce.html
- Hadoop Streaming -
  https://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html

Automated Processing

Automation is the natural step after identifying a high-value manual process. The process does not have to detect malicious activity. It can also create new datasets or add context to existing data, like joining VPN and web proxy logs to add a user id field to the proxy logs.

There are two types of automated processing: batch and streaming. Batch processes are generally scheduled to run at a certain time and read a "batch" of data. These processes read a batch of data from a disk backend, like HDFS or s3, process it, and store the results. In batch processing, there is always a time lag between data ingestion and processing.

Streaming jobs usually run continuously and consume a "stream" of data from a streaming backend, like Kafka or Kinesis. They often write data back to the streaming backend where another job consumes it, but can write data back to other data stores. A key difference in the streaming approach is processing data upon ingestion so lag time is near zero. This may sound appealing, but it brings significant challenges to address such as:

- Processing events exactly once, even in failure scenarios
- Monitoring streaming pipelines is more complex than batch monitoring
- Scaling components of the pipeline can be non-trivial

We recommend starting with batch processing and move to streaming if use cases need it. Many problems do not need near real-time results and are addressable via batch jobs that run every ten minutes or longer.

References for learning more

- Introduction to Kafka - https://kafka.apache.org/intro
- Kafka Quickstart - https://kafka.apache.org/quickstart
- What is Amazon Kinesis? - https://docs.aws.amazon.com/streams/latest/dev/introduction.html
- Getting Started with Amazon Kinesis - https://docs.aws.amazon.com/streams/latest/dev/getting-started.html

Response Activity

The set of response actions available generally does not change when an organization adopts big data. Common actions of sending an email or page, opening an incident ticket, or adding a rule to a security control point are low computation tasks not enhanced by a big data environment. However, there are areas where big data does change response activity.

Big data enables more analytic capability when an automated response requires manual analysis. With big data, we can efficiently process larger amounts of data to find contextual information about an incident. Given an alert for a suspicious DNS lookup by an IP address on the network, it is helpful to determine:

- the type of host associated with the IP address
- the user associated with the host
- which process made the request
- the sequence of DNS lookups before and after the suspicious lookup

This contextual information can expedite manual analysis and help identify earlier indicators of compromise.

You must also consider the impact of false-positives. Alert fidelity must be closely monitored in big data systems. The ability to process and alert on more data faster than before comes with an increased risk of overwhelming downstream systems and processes. Automatically inserting indicators from threat feeds into DNS alerting rules that auto-create cases can overwhelm an incident management tool and create a DoS incident when Facebook is inadvertently placed in a threat feed. It is important to build safeguards around automated detection and response systems to limit these scenarios. On the detection side, checking threat feeds against a list of benign domains before inserting them into a rule will help ensure this does not happen. Building automatic shut-off capabilities into the response system can prevent it from creating more cases if a certain amount of alerts fire for a single rule within a period of time.

System Improvements

Feedback mechanisms are critical for improving the effectiveness and quality of data environments. They allow users (or maintainers) of the data system to provide input on what is working and what is not. Hallway conversations and emails tend to get lost, so it is important to establish a clear process for submitting, reviewing, prioritizing, and implementing improvement ideas. Below are ideas to help assess several areas of system improvement.

Data Sufficiency

Data sufficiency is having the right data at the right time to answer a question.

The data must have enough detail or context to answer the question. Passive DNS data collected post-resolver will not enable client attribution as the IP address of the resolver is in the records, not the IP address of the originating clients. In this case, the data collection mechanism needs to be changed. As another example, sub-second timestamps are needed to order DNS queries. If the data ingestion process truncates timestamps to second-level granularity, there is a data sufficiency issue.

In other cases, a data set may need to be joined to another data set to add context. For instance, to analyze rates of incidents by asset type (laptop, data center server, etc), you may need to join incident data with data from an IP address management system.

Data must also arrive in time to be actionable. It is not useful to collect client attribution data (e.g. VPN logs) once a day and enrich web proxy alerts with attribution information every hour.

Quality Issues

Data quality is concerned with the validity and trustworthiness of data. Poor data quality can lead to incorrect conclusions, wasted response efforts, and the inability to identify compromised assets.

A common example of a data quality issue is a lack of time synchronization (e.g. NTP) which impacts the ability to order and time-bound events. There can also be issues with missing data, such as a field in a data set always being null or a data source that stops sending data for a period of time. Partial data can also be a problem and is common with UDP Syslog where there is a 1,024 byte limit on the packet [https://tools.ietf.org/html/rfc3164#section-4.1] that can result in truncated events.

Schema Issues

Schema issues relate to the structure of the data and often occur as a result of miscommunication between the data provider and consumer. Data sources adding or removing fields can break data pipelines or result in data sufficiency issues. Another schema issue is improper data formatting, such as quoting integers like Unix timestamps. This can cause processing to treat the data as characters instead of integers and impact operations like sorting.

Technology Issues

It is helpful to periodically assess the technologies you are using to ensure they are meeting your needs from both a data analysis and an operational perspective. Are you able to run the types of analysis you need? Does the technology scale to the size of your data? Are the technologies accessible to new users? Are they overly burdensome to administer? Are there new technologies that would be more beneficial for your organization?

Tools

Github (https://github.com/) or Gitlab (https://about.gitlab.com/) are useful in tracking feedback and issues.

Future Work

This document is intended to help organizations which have no background in big data to get started. It is also intended to grow over time as big data technologies change and as the SIG has more time to devote to it. Given that, there are several areas which should be included in future work:

- Cloud Computing - The majority of expertise in the SIG at the time of writing is for on-premise deployments though there are many capabilities offered by cloud providers such as Amazon, Google, and Microsoft.
- Streaming / Near Real-time analysis - Streaming data and incident detection within streams are advanced topics planned for a later revision.
- Taxonomies and Ontologies of IR data - Understanding the semantics of the data we ingest and process is becoming ever more important as we add automation to our processes. In the big data realm, this is only slowly taking off with platforms like ACT, but we will cover this in the future.
- Business Intelligence - Systems which connect to data stores and provide interactive querying, aggregations, and visualizations with little or no code required will be covered in a later edition.