



Practical Data Mining and Analysis for System Administration

Tanner Lund, Brigham Young University

Tanner Lund is a research assistant at Brigham Young University studying Information Technology. His fields of study include system administration and network management, with a specialization in distributed computing and log analysis. He has a strong interest in machine learning and applying its principles to network management.

Hayden Panike

Mr. Samuel Moses

Dr. Dale C Rowe, Brigham Young University

Dr. Rowe has worked for nearly two decades in security and network architecture with a variety of industries in international companies. He has provided secure enterprise architecture on both military and commercial satellite communications systems. He has also advised and trained both national and international governments on cyber-security. Since joining Brigham Young University in 2010, he has designed a variety of courses on information assurance, cybersecurity, penetration testing, cyber forensics, malware analysis and systems administration and published over a dozen papers in cyber-security.

Joseph J Ekstrom, Brigham Young University

Dr. Ekstrom spent more than 30 years in industry as a software developer, technical manager, and entrepreneur. In 2001 he helped initiate the IT program at BYU. He was the Program Chair of the Information Technology program from 2007-2013. His research interests include network and systems management, distributed computing, system modeling and architecture, system development, Cyber security and IT curriculum development.

Practical Data Mining and Analysis for System Administration

Abstract

Modern networks are both complex and important, requiring excellent and vigilant system administration. System administrators employ many tools to aid them in their work, but still security vulnerabilities, misconfigurations, and unanticipated device failures occur regularly. The constant and repetitive work put into fixing these problems wastes money, time, and effort. We have developed a system to greatly reduce this waste. By implementing a practical data mining infrastructure, we are able to analyze device data and logs as part of general administrative tasks. This allows us to track security risks and identify configuration problems far more quickly and efficiently than conventional systems could by themselves. This approach gives system administrators much more knowledge about and power over their systems, saving them resources and time.

The system is practical because it is more straightforward and easier to deploy than traditional data mining architectures. Generally, data analysis infrastructure is large, expensive, and used for other purposes than system administration. This has often kept administrators from applying the technology to analysis of their networks. But with our system, this problem can be overcome. We propose a lightweight, easily configurable solution that can be set up and maintained by the system administrators themselves, saving work hours and resources in the long run.

One advantage to using data mining is that we can exploit behavioral analysis to help answer questions about points of failure, analyze an extremely large number of device logs, and identify device failures before they happen. Indexing the logs and parsing out the information enables system administrators to query and search for specific items, narrowing down points of failure to resolve them faster. Consequently, network and system downtime is decreased.

In summary, we have found in our tests that the system decreases security response time significantly. We have also found that the system identifies configuration problems that had gone unnoticed for months or even years, problems that could be causing many other issues within the network. This system's ability to identify struggling devices by early warning signs before they go down has proven invaluable. We feel that the benefits of this system are great enough to make it worth implementing in most any professional computer network.

Introduction

System administrators are essential resources for almost every organization. Computer systems have become not only ubiquitous in modern businesses, but crucial for financial, marketing, sales, operations and other core business functionalities. System administrators are responsible for managing and maintaining a broad range of systems and services which are critical for core

business functions. These administrators are busy behind the scenes maintaining and monitoring the infrastructure that runs all the computer systems of a company. Because they have so many responsibilities and tasks to complete, it is important to make sure they have tools to help them keep track of all the different systems.

One of the many responsibilities of system administrators is to determine the needs in the network and computer systems for the organization prior to purchasing and configuration. Another responsibility is to install all network hardware and software and maintain it with the needed upgrades and repairs. They are in charge of system integration, making sure that all network and computer systems are operating correctly with one another. Because computer systems are largely opaque to regular observers, administrators need to collect systems data in order to evaluate network and system performance, to improve upon the systems to make them better and faster. They control the domain and are the primary people managing user accounts and access to managed systems. System administrators automate monitoring systems to alert them when issues appear. They are tasked with securing the network, servers and systems under their responsibility. They often train users on the proper use of the hardware and software that they are involved with¹. Some system administrators are tasked with the day-to-day help desk problems that users run into as well, like login issues or desktop computers breaking. It is difficult for system administrators to keep up with all the computer-related systems and tasks in an organization.

With all these different and demanding responsibilities on system administrators, it is important for them to know as much as possible about their systems and monitor their health. Administrators can spend a lot of time implementing different tools to try and understand their infrastructure, but there is a better way, a way which more effectively utilizes information already available to them. The key to improving the system is to implement a practical data mining infrastructure. System Administrators can use it to reduce wasted time and leverage existing infrastructure data and logs to easily manage many tasks.

Practicality of Data Mining Infrastructures

The average systems administrator has more tasks a day than he can accomplish in his limited amount of time. These tasks are usually prioritized by business needs and rate of completion. The more complex a task gets, the more time it takes away from other responsibilities. These tasks do not include the occasions when problems arise which constrain administrator time even further. Due to the nature of their work, a system administrator must quickly become friends with the data and logs inherent in the systems he maintains. Most of the time this means that an administrator is moving from system to system collecting the data he finds relevant. If the number of critical systems is too large to manage one at a time, then the administrator may employ some sort of tool such as Nagios or Zabbix to help do the monitoring for him. Setting up these tools takes time and maintenance, and it adds overhead to the administrator's daily tasks.

One often overlooked administrative tool is to mine the data that each machine creates by default. Much like the administrator who goes from system to system looking at individual logs, centrally collecting and data mining these logs can provide the same correlation more efficiently.

To understand what we mean by correlation, let us look at potential sources of data: All infrastructure environments include servers, workstations, and other network connected devices. Each type of device produces data in the form of logs, debug code, and often, authentication requests. The data in these examples are usually presented as a simple text message with the relevant information. A server will produce event logs from individual applications, as well as from its operating system and hardware components. This data is kept locally on the machine, but it can be forwarded to alternate locations for storage or further processing. Once this data has been generated and moved it can be processed to turn information into knowledge.

Using text based search, parsing, and retrieval, an administrator can process his data into units that provide understanding. Since data is being processed in near real time the administrator is able to compare multiple sources of incoming data all at the same time, in the same place. The administrator does not have to go individually to each machine to find information. Better yet, by correlating logs from multiple systems, the administrator is able to gain a greater understanding of how the action of one machine affects the entire ecosystem of his infrastructure. Equipped with a better understanding of his infrastructure as a whole, an administrator can spend more time preventing problems from happening, rather than just fixing problems after they happen.

Advantages of Data Analysis

Analyze Large Amounts of Data

Data aggregation systems are a part of this model precisely because they can handle much larger data sets. Servers, workstations, and network devices generate orders of magnitude more data than system administrators generally see about them, even with monitoring tools. These data aggregation tools allow the ingestion and analysis of much more than just SNMP traps. By collecting the log data of devices on the network and sending it to be processed and analyzed, knowledge of the network can be vastly increased.

Log data provides insights into almost anything a system administrator could care to know ². Every login, connection to a network, program error, overheating warning, and flapping port is reported in one way or another. The system we propose not only aggregates this data, but turns it into useful information and provides it to the system administrators in a more easily understood format. This approach can even be extended beyond device logs to services and programs themselves, enabling application-layer analysis.

Easily Identify Device Failures

One great advantage of having so much information to pull from is that it gives administrators the power to do predictive analysis. An excellent example is that of device failures, as devices often show signs of problems well before they die. Take this example log message:

Error Message \%\ASA-3-210007: LU allocate xlate failed for type [static | dynamic]-[NAT | PAT] secondary(optional)protocol translation from ingress interface name : Real IP Address / real port (Mapped IP Address / Mapped Port) to egress interface name : Real IP Address / Real Port (Mapped IP Address / Mapped Port)

This message indicates a memory allocation failure, likely due to all available memory being allocated or memory corruption. Either way it is a problem that should be investigated. There are hundreds of log messages similar to this generated spontaneously throughout most networks. They can report memory errors, power failure errors, network and connection issues (like port flapping, or lots of dropped connections), and many other unfavorable changes. We have found through first-hand experience that without a proper log analysis engine, these logs often go unread and these issues go unnoticed for months, sometimes even years. It is impossible for system administrators to manually read every log from every system on a daily basis. Log analysis can help catch these problems early.

Explicit Behavioral Analysis

This concept of behavioral analysis can also be applied to cybersecurity incidents. Certain log patterns are left behind by most types of cyber-attacks, for example: attempted root logins, high volumes of network traffic, and periodic mass-disconnects from access points. Some attacks are explicitly mentioned in the logs generated by network devices, and it is also possible to forward logs from an IDS into the system. Even attacks that don't leave such an obvious trail can be detected by correlation of the patterns they leave behind.

By using this system, we have identified the patterns left behind by a number of different attacks. As we have done so, we have been able to modify our data filters and alerting models in order to watch for these patterns to repeat. We shall summarize some of our findings hereafter.

Data Gathering System Components

There are five common components found in most data analysis systems. These include:

- “Shipper” programs, which send logs from the devices to a central location
- “Broker” programs, which aggregate the device data and hold it until ingestion, often in a queue or a designated directory

- “Indexer” programs, which break the data out into searchable fields
- “Search and Storage” systems, which receive the indexed data and store it, also providing the ability to search over the data according to the indexed fields.
- The interface, often a web interface, which provides tools and views that enable users to search and make the data meaningful.

This event cycle³ reflects the processes involved in turning events and their logs into useful information, regardless of the implementation.

Further System Explanation

The core of this methodology utilizes search or data mining technologies. There are many tools that provide the ability to analyze large amounts of data and produce usable information. Tools currently available include: Splunk⁴, Apache Solr, and Elasticsearch Foundations’ ELK stack (Elasticsearch, Logstash, and Kibana). Each of these tools has the ability to take textual data and parse it into desired fields of information. For the purpose of this paper, we will be using the ELK stack as our software of choice.

Our practical data mining system includes: data sources, a shipper, a broker, an indexer, searchable storage, and an interface. Each piece serves as a crucial part of an integral system. A piece of data enters the system directly from the data source, or a shipper, from which point it is placed in a broker. The indexer pulls the data from the broker, and the data is parsed and placed into the searchable storage. The interface then accesses the storage through API calls. This same basic concept is used by most data mining implementations.

Let us begin with our data sources. Data sources can be anything that produces text-based events. Common log sources include: servers, workstations, printers, and network equipment (such as wireless access points or routers). In reality, we produce data events in everything we do with computer systems⁵. Each time we browse a web page, purchase an item, or a click on an ad, we produce events. Things that we would normally not consider data sources often produce vast amounts of data. For example, HVAC units produce a considerable amount of useful data that can be leveraged to better control the climate of a server room. Again, any system that produces textual data can be a potential source. Once we have identified our potential data sources we must get the data from that source into our shipper, or directly to the indexer, so that we can glean useful information. For the purpose of scalability, resilience and performance, we suggest that data be sent through a shipper.

The shipper can be thought of as a cargo ship. Many different data sources can be configured to send their information to the shipper. The shipper, in this case Logstash, allows us to have some sort of redundancy between the original source and the rest of the cluster. In this manner, if an organization were to lose an indexer, data is not lost. The shipper allows some preprocessing on

the data it receives. The shipper performs basic tagging and sorting of different classifications of data. For example, if we find that a piece of data came from a Cisco device, we can tag that piece of data with a keyword that will allow for quicker downstream processing. Using preprocessing on the shipper also allows for decreased loads on the indexers. The shipper can be configured to send directly to the indexer. Better yet, the shipper can be configured to send the partially processed data to a broker.

The broker is able to absorb any fluctuations that may occur upstream at the data source. If a data source begins to experience problems, the amount of data it produces may increase exponentially. If all this traffic is pointed straight to an indexer, with no shipper and broker buffer, it is entirely possible to overflow the processing abilities of the indexer. The result is that the cache on the indexer runs out of space, and therefore data will get dumped on the floor. In critical situations, every piece of data is valuable. To overcome this, brokers provide storage in the form of a cache. Many different types of brokers can be utilized. For the purpose of this paper, we suggest using Redis as a key value store. The shipper sends its partially processed data to the Redis buffer. While not fully developed, Redis has the ability to be clustered with replication. When clustered, in theory, Redis should provide complete lossless caching. This provides a large broker cache to absorb those large data surges. Depending on the desired configuration, Redis can be a simple cache from which the indexers pull, or it can be configured with publish and subscribe functionality. In either situation, Redis as a broker can help guarantee the stability of a data mining infrastructure.

The indexer, in concept, takes provided data and breaks it apart according the rules specified by the user. As discussed above, the indexer can be configured to receive data directly, or it can pull the data from a broker. Different indexing software accomplishes the data parsing in different ways. The ELK stacks uses the Apache Lucene language as its base for text based parsing and retrieval^{6,7}. The actual parsing of the text logs is accomplished in the Logstash portion of the stack. Logstash uses the Grok library to match patterns in text. Grok is an open source natural language processing library that uses regular expressions saved in the form of variables⁸. By using certain Grok patterns we map them to the known format of data sources. Many formats have been pre-written into the Grok library, including general syslog, specific syslog formats, and many Cisco formats. If the format has not yet been included in the Grok library, adding new regular expressions is straight-forward. The parsed data is then passed into the searchable storage database in the form of a JSON object. This object notation allows for easy searching of the parsed fields, and all the original data is retained with the JSON.

The searchable storage portion of the ELK stack is provided by Elasticsearch. Elasticsearch uses indices in which the parsed JSON objects resides. The indices can grow to be very large, so it is common to have a new index created every day. If the number of indices grows too large, or the size of the indices grows too large, it becomes highly resource intensive to search the data. For this reason, programs such as Elasticsearch Curator are used to control the number of indices that

are opened and searchable. As with any storage system, a need to expand will always arise. To accomplish this, Elasticsearch breaks apart the storage into the roles of master, backup master, data node, and balancer. The indices are then shared across multiple data nodes to provide a more fault tolerant platform. Additional nodes of all roles can be added or removed with ease, allowing for simple horizontal expansion and robust failover. With the ability to quickly expand and adapt to changing needs, the Elasticsearch storage provides great stability and scalability⁹.

With a stable storage platform, we need something to access the data residing inside it. In the ELK stack, Kibana provides the data interface through which data is analyzed and correlated. Kibana is a web front end, similar to Splunk. Kibana uses API calls to the Elasticsearch storage to sift through parsed data using user created queries. These queries utilize boolean logic to help string together complex searches. By stringing together multiple ideas over multiple data sources, we can correlate events that may have escaped an administrator's notice.

Architecture Overview

Having discussed a theoretical overview of a practical data mining system, we can now review the concrete examples of how a system is configured. We will be using the Elasticsearch ELK stack as our mining system. For detailed instructions on how to get the ELK stack installed on a chosen distribution, please refer to the Elasticsearch documentation. This documentation can be found here: <http://www.elastic.co/guide/en/Logstash/current/Logstash-reference.html>

The ideal architecture of a practical data mining system for systems administrators should contain shippers, brokers, indexers, storage, and an interface. While it is possible to forgo the use of the forwarder and broker, we suggest both be employed to create a more robust system. By using a shipper and broker combination, the flow of data can be redundant and resilient.

In an ideal situation, a shipper will sit on every data source, or there should be an aggregated shipper designated for the data source. Using a Logstash shipper requires Java. A shipper preprocessing large amounts of data can cause significant resource utilization. If resources are an issue, it is possible to use any number of open source shippers available publicly on GitHub. Some potential third party shippers include Logstash-forwarder, Beaver, and Woodchuck. Not all of these third party shipper support the use of Redis as a broker, so care must be taken in architecting the redundancy of the system.

A broker should be implemented. In this example, we have used Redis as our broker, but there are several other options. Some of these options include the use of AMQP or RabbitMQ. Regardless of the chosen software, a broker will help to absorb fluctuations in the flow of data. Since the broker will cache all incoming data, no data will be lost if an indexer cannot keep pace. We suggest using the broker in some form of cluster. Using clustering and replication will help to ensure resilience within the distributed broker cache. While we have yet to implement a

complete publish-and-subscribe infrastructure, our research has demonstrated a need for this model. With high processing loads across multiple categories, a publish-and-subscribe environment can help segregate the workload.

To help ensure the resilience of the infrastructure, we suggest utilizing multiple indexers. In a publish-and-subscribe environment, there should be at least two indexers per class of data. If the publish-and-subscribe method is not used, care must be taken to ensure there are sufficient indexers to handle the workload even during peak throughput. To have enough resilience in the system, it is always a good idea to have $n+1$ indexers. With the use of a broker, it is possible to lose all but one indexer for each class of data with pub/sub, or all but only one indexer for operating the broker as a simple key-value store. By keeping at least one indexer alive, data can still be processed into the searchable storage.

Once all the data has been funneled and processed, it must be stored in some sort of searchable fashion. We elected to use straight Elasticsearch for the searchable storage. There are other options, including es-hadoop, but we found straight Elasticsearch to be the simplest to set up and scale.

Elasticsearch breaks the searchable storage component into 3 roles: master nodes, balancer nodes, and data nodes. Each role serves the function for which it is named, and all roles should be assigned for the best functioning system. Each role has different resource needs, but only the data nodes need serious design consideration.

We have found that it is ideal to have at least three master nodes. One master will be the active master, and the other two will provide backup. When there are only two master nodes they can sometimes fight for control of the Elasticsearch cluster. While this is not always the case, or even common, we would suggest the use of three masters to allow a majority vote which will prevent this problem. Master nodes tend to be heaviest on the CPU, so it is a good idea to allocate CPU resources accordingly.

The balancer nodes, also known as client nodes, serve the purpose of accepting data as http traffic. When a query comes in, the balancer nodes will help direct the traffic to the correct location. Since the balancers handle all the http traffic, the data nodes are freed up to do the heavy processing of the queries. It has been our observation that balancer nodes do not seem to use many resources, though in extremely large environments, network and CPU utilization may be strained. It is always a good idea to have at least one balancer node. In larger environments it is wise to have multiple balancers.

The final Elasticsearch role is that of the data node. The data node is arguably the most important role. The data node has many tasks including storage, sharing, and querying. We have found the data nodes to be the most resource intensive. These nodes can consume significant amounts of

RAM, disk, and CPU resources. As it is easy to add new data nodes, we suggest that additional extra data nodes be used for horizontal scaling, rather than by vertically scaling a single node. In the end the majority of nodes will be data nodes. By combining all of the Elasticsearch roles, it is possible to create a robust, efficient, searchable, data storage cluster.

With the data stored in the Elasticsearch cluster, we now need to present the data in a useable fashion. To present the data we used Kibana. Kibana is a web frontend that runs on multiple server platforms. The Kibana interface needs access to the Elasticsearch cluster, so it is usually a good idea to install it on a balancer node. Using the Kibana interface, a system administrator can submit powerful queries to searchable Elasticsearch storage. It is also possible to interface directly with the Elasticsearch API. This can be helpful when choosing to write custom alerting and data modeling applications. Elasticsearch provides the percolator API to allow for alerting and monitoring. By setting up the alerting, we have completed the robust system needed for practical data mining.

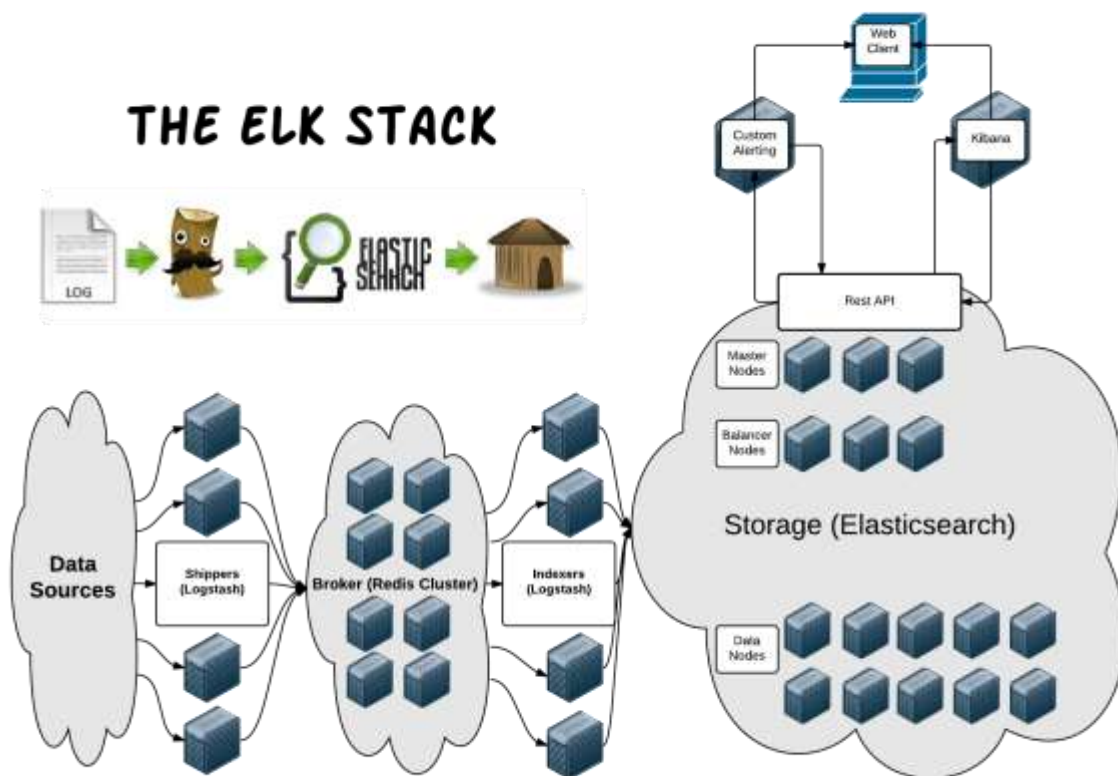


Figure 1 - Ideal Data Mining Infrastructure Architecture

Figure 1 is an ideal data mining infrastructure for systems administrators. The number of nodes for each service should be changed to fit individual needs. As discussed above, the parts of the system can be switched out, but we found this to be a very effective solution.

Data Source Configuration

Different data sources require different methods for collecting the data. As aforementioned, when at all possible a shipper should be installed on the data source. If this is not possible, most servers and networking hardware support some versions of syslog. Syslog can be configured to forward logs to external locations. As an example, rsyslog can be configured to forward all logs with this command:

```
*.* @@<hostname or ip>:<port>
```

Shipper Configuration

Logstash provides a basic shipper composed of three basic parts: an input, a filter, and an output. The shipper will take an input, usually directly from the log files. Depending on the location of the file, the Logstash user may need to be part of a group that has access to the file. Some preprocessing can be done in the filter section. This preprocessing usually involves tagging a piece of data with a type. The type can be used later to speed up the filter matching process. Finally, the output portion is structured to send the partially processed data to the broker.

In the example below, we have set the path to the data file as being /var/log/messages. The default behavior of Logstash is to treat a file like a stream. Since it is treated like a stream, Logstash starts reading at the bottom and does not read anything above it. In this case, we are assuming that there already exists valuable data in the log file. For this reason we specify the start position as the beginning. This forces Logstash to read everything currently existing in the file. We have also tagged the type as syslog. When we look at the output we will see why tagging the type is important. Since all we need for preprocessing is to set the type, there is nothing in the filter portion. The output is specified to ship to the Redis broker. There is a slot for specifying the Redis hostname or IP address. We have also specified that the data_type be a channel and not a list. By changing the data_type to channel, we have specified that Logstash use PUBLISH instead of RPush. Finally, the key is a compound key. It contains the word “Logstash”, combined with the data type and the date. The output also contains a stdout debug line. This is there to help verify that Logstash is actually reading the file correctly.

```
input {  
  file {  
    path => "/var/log/messages"  
    start_position => beginning  
    type => "syslog"
```

```

    }
}

filter {
    <More preprocessing if wanted>
}

output {
    Redis {
        host => <hostname>
        data_type => 'channel'
        key => "Logstash-%{@type}-%{+yyyy.MM.dd.HH}"
    }
    stdout {
        codec => json
    }
}

```

Broker Configuration

With the shipper working, the keys should be visible in the Redis cache. The keys can be shown from the Redis host command-line with the command *KEYS *Logstash**. This will show all keys with Logstash in them. By default, the Logstash indexer constantly pops the top of the list format, keeping the memory in a steady state. However, in our model we are using channels rather than lists. When a Logstash indexer pulls data from a channel, the data isn't removed, causing growth. As the data volume grows, memory management is required. To accomplish this, the default Redis configuration file, usually */etc/Redis/Redis.conf*, should be modified to change the max memory and the max memory policy.

The *maxmemory* line tells Redis to stop increasing cache once it hits 500mb. In theory, this can be set to be much larger, but indexers should be able to keep pace with the 500mb limit. In the *maxmemory-policy* line, we are setting Redis to drop on "least recently used." This should drop the data with the oldest timestamp in the key, in essence setting up a first in first off stack. It is possible to take extra steps to clustering Redis with replication, but for the sake of this setup we will leave it be.

```

maxmemory 500mb
maxmemory-policy allkeys-lru

```

Indexer Configuration

Very similar in configuration to the shipper, the main differences for the indexer are the location the data is pulled from, the way the data is handled, and the output location. We still have the three part configuration with an input, filter, and an output. It just needs to be specified so the now parsed data can make it to the searchable storage.

In the input section, we are looking to pull from the Redis cache. We specify the input protocol as being Redis. The host will be name of the Redis host or cluster. For the publish-and-subscribe method we need set the `data_type` to `channel` or `pattern_channel`. As can be seen in the example below, we will be using `pattern_channel` for simplicity's sake. The `pattern_channel` data type tells Logstash to gather the data using a `PSUBSCRIBE`. This means that Logstash will subscribe to any channel that has a key that contains the pattern specified. In the example below, we have specified that key to be "Logstash". In the shipper we set the key to always have the word "Logstash" in it, so the `pattern_channel` will subscribe to all channels. To further break up the workload, the indexer can be setup with the `channel` data_type, and the key can be specific to the desired data type.

Now that the data has reached the indexer, we want to do the heavy parsing. In the shipper we tagged the data with the type of syslog. The first thing we want to do is make sure we are looking at a piece of data that has been tagged with syslog. If it has been tagged with type syslog, we want to allow Grok to parse the data with the preset `SYSLOGBASE` pattern. This is a prebuilt pattern, but can be customized for different types of data. This Grok will spit out a JSON with all the appropriate syslog information, along with the message.

Now that we have a parsed JSON, we want to store this information in the searchable storage. The output configuration is different this time, in that we will be pushing the data into the Elasticsearch cluster. We have chosen to use the Elasticsearch protocol, and we have specified the data to be sent to the designated Elasticsearch cluster. Since we did not specify an IP address, the cluster is discovered by means of multicast broadcasting. If the cluster name matches, then the data will be pushed. It is usually a good idea to have some sort of rotating index which prevents one index from growing too large in size. Here we specify the index be `Logstash-date`. This will help make sure that a new index is created for each day. In a large environment it is a good idea to name indexes according to the type of data they hold. Shown below are a sample indexer configuration, a sample log, and the resulting JSON output.

Configuration:

```
input {  
  Redis {  
    host => <hostname or ip>  
    data_type => "pattern_channel"  
    key => "Logstash"  
  }  
}
```

```

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGBASE}
+ %{GREEDYDATA:message}" }
    }
  }
}

output {
  Elasticsearch {
    cluster => <cluster name>
    index => "Logstash-%{+YYYY.MM.dd}"
  }
  stdout {
    codex => json
  }
}

```

Original Log:

Mar 14 23:40:42 BoBv3-Ubuntu kernel: [26421.152706] ACPI Error: Method parse/execution failed [_SB_.PCI0.LPCB.EC0._Q0E] (Node ffff8802264491e0), AE_AML_UNINITIALIZED_ELEMENT (20140424/psparse-536)

JSON Output: (see Figure 2)

```

{
  "STELCHBASE": {
    "Mar 14 22:40:42 SoReS-Cluster base1/"
  },
  "timestamp": {
    "Mar 14 22:40:42"
  },
  "MONTH": {
    "Mar"
  },
  "MONTHDAY": {
    "14"
  },
  "TIME": {
    "22:40:42"
  },
  "HOUR": {
    "22"
  },
  "MINUTE": {
    "40"
  },
  "SECOND": {
    "42"
  },
  "STELCHFACTORY": {
    null
  },
  "facility": {
    null
  },
  "priority": {
    null
  },
  "logsource": {
    "SoReS-Cluster"
  },
  "IPVHOST": {
    "SoReS-Cluster"
  },
  "HOSTNAME": {
    "SoReS-Cluster"
  },
  "IP": {
    null
  },
  "IPV6": {
    null
  },
  "IPV4": {
    null
  },
  "SYNLOGMSG": {
    "kernel"
  },
  "program": {
    "kernel"
  },
  "pid": {
    null
  },
  "message": {
    "
    * [24471.132764] ACPI Error: Method parse/execution failed [\\_SB_.PCI0.LPC
    CR.RCS _Ox] (Node f2f18d3284481e0), AE_AML_UNINITIALIZED_ELEMENT [20]60434/pa
    rams-538"
  }
}

```

Figure 2 - JSON Output

Searchable Storage Configuration

While the Elasticsearch configuration can seem like the most daunting part, implementation is actually comparatively simple. We will first set some of the basic parts in the `/etc/Elasticsearch/Elasticsearch.yml`. The first change will be setting up a cluster name.

The cluster name must be identical across all the Elasticsearch nodes as this is used by the aforementioned multicast discovery method. To set the cluster name, the `cluster.name` property is updated. By default this name is set to `Elasticsearch`.

cluster.name: <new cluster name>

Now that we have changed the cluster name, we want to change the node name. This will help us to distinguish between the different nodes in the cluster. The name change is found in the same file as property `node.name`.

node.name: <new node name>

The next basic configuration is for node type. The node types can be broken out into the three different roles, the master, balancers, and data nodes. Two lines in the `/etc/Elasticsearch/Elasticsearch.yml` specify what type of role a box will be. We will be looking at the lines with `“node.master:”` and `“node.data:”`. By default all nodes are considered to be data nodes, unless the value of `node.data` is changed to `false`. If `node.master` is `true`, then this node will be eligible to be a master node, and cannot be a data node. If both master and data are `false`, the node becomes a balancer or client node.

node.data: <true or false>

node.master: <true or false>

By completing these 3 basic steps, the Elasticsearch cluster is ready for use. There are many “advanced” settings, but for the purpose of getting a basic cluster up, we have succeeded. For tips on more advanced Elasticsearch settings, please refer to the advanced tips section.

Interface Configuration

There are many different interfaces, though Kibana is the most prevalent. Elasticsearch has recently made the transition to Kibana 4, which comes with its own built in web server. Setting up Kibana can be a rather laborious process, and with the major changes that came with Kibana 4, it is best to refer to the company’s tutorial. Kibana4 is arguably the most powerful portion of the ELK stack, with many built-in features for data querying and modeling.



Figure 3 - Example Kibana 3 Interface (source Elastic.co)

The two interfaces are displayed in Figure 3 and Figure 4 (credit elastic.co).



Figure 4 - Example Kibana 4 Interface (source Elastic.co)

Tips and Optimizations

Grok Patterns

Grok uses regex patterns and assigns them two variables. Hierarchical nesting is inbuilt allowing a Grok that is made up of smaller Groks. As regex Grok patterns can be difficult for new users, it is handy to test them out. We found the website <https://grokdebug.herokuapp.com> to be very helpful. There are options to attempt a discovery, pre-existing patterns, and new debug patterns against a real block of text.

Shippers and Indexers

In this example, both the shippers and the indexers were built using basic Logstash. Logstash will read any .conf file placed into the Logstash/conf.d folder. This means filters can be organized between several files. This can aid in debugging, when the .conf files are named by their purpose.

With only a few shippers and indexers in a large environment, each can be seen with many inputs and outputs. Often times the inputs and the outputs use different protocols, and those protocols may or may not overlap. We found that it is simpler to have many different boxes that each support one or two inputs and a single output. This makes setup straightforward. We advocate using Redis as a broker. Redis acts as a great middleman, and it helps to eliminate overhead.

Searchable Storage

The first thing we can do to optimize performance is to increase the Elasticsearch jvm heap size. This setting is found in /etc/defaults/Elasticsearch on Debian. In this file, we are specifically looking for the line with ES_HEAP_SIZE. This should be uncommented and changed to be no more than half the total RAM in the system. Elasticsearch also advises not to give above 32G of RAM to the JVM as once this is exceeded, the JVM can no longer compress pointers to 4 bytes per pointer. Instead all pointers will be 8 bytes, causing decreased ROI. Elasticsearch has made it easy to scale horizontally, and we recommend this approach.

ES_HEAP_SIZE=<number>G

When using a Linux box, we found significantly increased performance could be achieved by disabling swap for the Elasticsearch JVM. This is simply accomplished using the bootstrap.mlockall line in /etc/Elasticsearch/Elasticsearch.yml. Mlockall will lock the process address space in RAM. The mlockall command is best used in conjunction with the max_locked_memory command and should be set to unlimited in the /etc/defaults/Elasticsearch file. This will give Elasticsearch permission to set its own memory.

/etc/Elasticsearch/Elasticsearch.yml
bootstrap.mlockall: true
/etc/defaults/Elasticsearch
MAX_LOCKED_MEMORY=unlimited

To optimize large, diverse queries, the field cache should be limited. The field cache stores queried field data for quick retrieval. As data begins to increase in its diversity, the field cache grows, and the stored data may not have what is being sought for. We want to limit the cache so that we have memory available for newer queries. By default, the field cache is unbounded, but

we have found that it is best to limit it to 30% or 40% of JVM heap size. This setting can be added to the `/etc/Elasticsearch/elasticsearch.yml` file under `indices.fielddata.cache.size`. Previously, we had used the cache expire setting, but Elasticsearch has stated that this value may soon deprecate.

indices.fielddata.cache.size: <Percentage or fixed value>

Elasticsearch has the ability to keep multiple replica shards for failover purposes. By default, there are around 2 replica shards per index. As a cluster size grows, the number of replica shards should be increased. Replica shards help a lot when the cluster need to balance, or if it is recovering from a failure. We currently run 5 replica shards. This setting can be changed with a curl command, or it can be set in an API call.

CURL

```
curl -XPUT '<hostname>:9200/my_index/_settings' -d '{
  "index": {
    "number_of_replicas": <number>
  }
}'
```

/etc/Elasticsearch/Elasticsearch.yml
index.number_of_replicas: <number>

Interface

Kibana has the ability to take a basic set of data and turn it into a gold mine. There are many different types of ways to model data as simple queries can provide really interesting answers. Note that in Kibana 3, changes are always lost if queries are not manually saved before reloading.

Data Sources

The first flows of data into an ELK setup can feel rewarding. However, as experimentation is increased, there is a natural desire to increase the amount of data collected. It is usually at about this point that the thought becomes *collect all data possible*. We have found that a lot of the data is just noise and is worthless. We suggest filtering the incoming data, so that only the most valuable data is collected.

Deployment Capabilities

Our deployment consisted of 10 surplus desktop computers of the following specifications running ELK roles on Debian 7:

- 2nd Generation Core i7 PCs
- 16 GB RAM
- 150 GB HDD

In terms of performance, this deployment was able to ingest over 1.1 million log events per minute consisting of wireless, network, IDS, virtualization, operating system and application logs. The system was deployed across a campus infrastructure supporting over 40,000 users.

A similar infrastructure (or one with similar capabilities) would be fairly inexpensive for even most modest organizations to implement. The cost of this system is minimal, especially in light of the value the system brings.

Findings/Usage Examples

Increased Security Response Time

In our network we found unauthorized VPNs and attempted internal hacks, which we were able to identify and shut down quickly. Some of the security incidents detected would not have been caught at all without this system, despite a well-trained cybersecurity team.

Some examples include:

- Xirrus generated IDS event logs (rogue APs, unauthorized connections, etc.)
- Large numbers of failed login attempts
- Deauthentication attacks (large numbers of simultaneous disconnections)

In the above examples, it was possible to use knowledge of Wireless AP locations to internally geo-locate attackers on campus. This can be enhanced with the mapping functionality built into Kibana.

Identified Configuration Problems

In large networks, it is inevitable that devices will be misconfigured or take on incorrect settings in some way. Sometimes these misconfigurations are immediately obvious to network engineers, but others are not. We identified port flapping, incorrect NTP settings, and incorrect time zones, among other configuration mistakes.

Identified Struggling Devices

We were able to identify many device errors and other signs of struggling devices. Log messages mentioned the following physical issues:

- “MALLOCFAIL”
- “IAP association table full”
- “memory inconsistency detected”
- “unable to establish tunnel...due to low system resources”
- “signal on power supply A is faulty”
- “memory fragmentation”
- “no alternate pool”

All of these messages related to hardware errors of one type or another. Most are memory errors. These may not keep a device from functioning, but they may hurt performance and indicate that there could be problems in the near future. A good example is the power supply problem. Clearly, there is a functioning secondary power supply, but when it dies the device will fail unless something is done now to fix the existing problem. The ability to proactively identify and address these kinds of problems can prevent service issues and outages, along with their associated support costs and user frustration.

Wireless Load Balancing

Access to wireless access point logs allowed for an entirely different form of analysis. Xirrus access points generate log messages for each connect and disconnect it experiences, as well as MAC address information for each connecting or disconnecting device. This data allows us to:

- Identify access points with a high load
- Identify access points with little or no load

With this information, a cost-benefit analysis can be made of the access points in a building. This allows the engineer to recognize where more should be added and where others are ineffective. As a result, this would improve coverage and reduce costs. In our deployment, we were able to recommend re-location of several access points to increase effective coverage rather than purchase additional hardware.

Tuning Recommendations

In light of these findings and our experience in working with this data analysis architecture, we have put together some general practice recommendations. We hope that they may help system administrators more effectively implement their own systems. These suggestions are vendor agnostic, but not all of them apply to every technology.

Many data analysis programs, particularly those which are free, do not auto-detect the fields that the data should be split into. Thus, an appropriate configuration will not be provided for the parsing system the first time around, unless the incoming data is already well known and a configuration can be pre-tailored. The best procedure is to build an initial throwaway cluster with minimal hardware or in a VM, tuning the configuration until it is satisfactory before deploying a production environment. Configuration in such systems is inherently iterative.

As problems are identified and resolved, we recommend that the process for identifying them be recorded. In the future, as events repeat, it will be easier to find and fix familiar problems. Additionally, this lays the groundwork for an alerting system of choice to be implemented. Along these same lines, we recommend maintaining a repository for useful Grok patterns, configuration information, devices managed, logs dealt with, etc. This could be a wiki or any other type of shared knowledgebase.

Future Work

In the future, Linux, Windows Server, and workstation logs should be integrated and assessed for their value in such a system. We believe that these logs can help in analysis and management of kernel, service, software versioning issues, and application layer root cause analysis, in addition to the benefits we discovered about log messages in general (misconfiguration, memory errors, etc.). An assessment of these logs would provide a more complete picture of this system's value to system administrators.

Further analysis can be done on access point data. If connections can be associated with disconnections, then deeper studies of wireless usage can be pursued. It will be clear which access points endure a large load at all times versus those which have the occasional spike. Additionally, the noise created by users who are travelling and thus connect and disconnect frequently (without using much data) can be separated from more valuable data. Wireless networks become much more efficient when user behavior can be examined in full.

This type of system has the potential to evolve into a much more powerful framework, allowing system administrators to manage even larger networks than before. As previously mentioned, the current configuration process is iterative by necessity. We hope to see implemented machine learning algorithms and configuration management systems, resulting in a self-managing network. This will allow system administrators to do a higher level of work, thus taking network management to a higher level of abstraction. General improvements in the data model and pattern-identifying methodology will also improve efficiency.

Conclusions

Our implementation of a data analysis system has proven very effective at identifying and facilitating the resolution of potential system issues quickly, including those that had gone undetected for some time. It provides greater insight into network behavior and extracts value from existing data that is often difficult to analyze in large quantities. Most importantly, the system was straightforward to deploy, use, and scale, which is a must in organizations without significant IT resources.

Considering the many benefits such a system provides, as well as its complexity as a tool, we have concluded that it is very reasonable to implement a data analysis architecture, and should be implemented in all but the smallest networks. Doing so will give administrators greater power over and insight into their networks and allow them to make more meaningful use of their time. As networks and their demands (including reliability demands) increase in size, this is not only useful but *necessary*. We encourage future system administrators and network engineers to develop a working knowledge of such systems and an understanding of their value in the modern world.

Bibliography

1. US Department of Labor - Bureau of Labor Statistics. Network and Computer Systems Administrators. In: Occupational Outlook Handbook. Bureau of Labor Stastics.
2. Eaton I. The Ins and Outs of System Logging Using Syslog. SANS Information Security Reading Room. 2003.
3. Elastic.Co. What is Elasticsearch. 2015.
4. Zadrozny P, Kodali R. Big Data Analytics Using Splunk. Apress; 2013.
5. Schneier B. Data and Goliath. W. W. Norton & Company; 2015.
6. Konchady M. Building Search Applications: Lucene, Lingpipe, and Gate. Oakton, VA: Mustru Publishing; 2008.
7. Gormley C, Tong Z. Elasticsearch: The Definitive Guide. O'Reilly Books; 2015.
8. Obermeier KK. GROK — a knowledge-based text processing system. In: CSC '86 Proceedings of the 1986 ACM Fourteenth Annual Conference on Computer Science. New York, NY: ACM; 1986. p. 331–339.
9. Turnbull T. The Logstash Book. 1.4 ed. www.Logstashbook.com; 2014.