Detecting System Log Loss Through One-Way Communication Channels

Jason Leverton

Detecting System Log Loss Through One-Way Communication Channels

GIAC (GCDA) Gold Certification

Author: Jason Leverton, Leverton.jd@gmail.com
Advisor: Clay Risenhoover

Accepted: November 11, 2020

Abstract

Organizations are consolidating log collecting, monitoring, and incident response activities. There are many reasons an organization could find itself in this situation, whether they are attempting their first deployment of security architecture or they are shifting to a SaaS Cybersecurity product. These data collection points may not always be located within the same trust boundary, or even within the same organization. They may also be communicating through highly restrictive gateways. These collection points could gather information from multiple networks, all with different classifications, security postures, or network owners. There are incidents when communication flowing from one organization to another may have restrictions on two-way communication and rely entirely on a one-way communication channel. The lack of a two-way connection presents a challenge when continuous monitoring is required. Most host-based agents and log transfer mechanisms rely solely on established connections (TCP). This paper examines the transfer of logs through a one-way communication channel. It aims to detect and measure the amount of log loss on the channel and intuit the time, size, and volume of log messages lost. The goal is not to provide error correction but instead to introduce error detection.

1. Introduction

The comparison of User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) is standard in any topic or course referencing the passage of information between two systems using a computer network. The comparison usually breaks down to a matter of reliability. The primary attribute of UDP is that it offers no assurance that the packet can be delivered to the end host. TCP provides the minimum assurance that the system at least received the message (Gerhards, 2008). There are many other differences, but this example provides a basic contrast between these two protocols. There are times that the flow of logs depends on UDP, and the implication of using UDP comes with some trade-offs. RFC 5426 is an RFC dedicated to the transfer of Syslog messages and (Okmianski, 2009) establishes the initial conditions, security and reliability consequences of utilizing UDP in a logging environment. Many articles written on UDP and logging focus on the engineering efforts that can be made on the network to reduce UDP logs' loss. Their content is a good starting point to design the network and ensure that UDP gaps are mitigated before the transmission of logs occurs (Syslog-ng, 2019). Many tools, guides, and best practices all point to the recommended method of using two-way communication channels with encryption when building a logging architecture. (Todd, 2017).

Stateful connections offer many advantages, such as reliability, flow control, and encryption. However, there are times when this is not possible. Some examples of situations that may require the use of one-way communication methods:

- Industrial Control Systems (ICS)
- Restrictive government or corporate enclaves
- Internet of Things (IoT) components with limited functionality
- Legacy networks and appliances

Some of these networks in the past have relied heavily on air-gapped network designs. The definition of air-gap is changing, as there are growing attacks. As mentioned in a recent newsletter, "Air-gapped networks are just networks with really high latency" (Ullrich quoting Skoudis, 2020). The failed use of an air-gap was also demonstrated notoriously by Stuxnet in 2008. (SANS ICS Blog, 2019). There are also instances when older technology does not offer the option of using TCP, or perhaps UDP Syslog is the mechanism to transport some IoT devices' logs.

As discussed, there are times when UDP offers an advantage but there are also large pitfalls while using UDP. The system presented in this paper may not offer data correction on the fly. However, detection of log loss and the introduction of alerting on log loss can increase early

Jason Leverton, Leverton.jd@gmail.com

indication of compromise and move closer to mitigating some of the inherent weaknesses with one-way communications. This idea of interacting with the log messages has been explored by creating messaging queueing services such as Kafka and RabbitMQ. (Ger, 2017). The idea of message queuing is certainly robust, and the system's inherent nature requires that all devices raise the communication bar to two-way communications.

The focus of this paper is the detection of log loss through a one-way communication channel. Two log aggregators are set up as a sender and receiver. The sender is only able to craft and send UDP packets to the receiver. A data structure inserted into the log channel is used to enumerate and alert on log loss. The results will be measured against an out of band comparison between the sender's log files and the receivers log files. The use of the diff command contains the exact log messages that were lost in transmission (Linux Manual Diff man page, 2020).

1.1 Network Setup

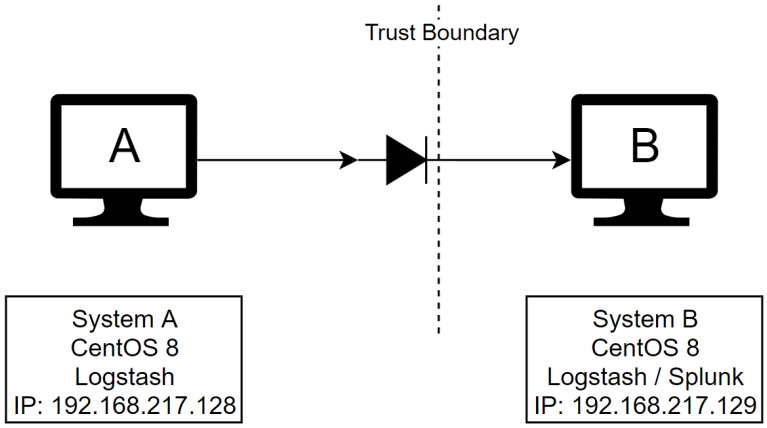

Figure 1: System to System Connection

1.1.2 Host Details

A single virtual environment contains all hosts and networks. Each virtual machine runs CentOS 8 64-bit running on VMWorkstation Pro v15.5.

1.1.3 Network Details

The virtual machines connect to a shared LAN segment. This network allows the devices to communicate only with each other and with no outbound internet connection once the initial configuration is complete. The machines on the network 192.168.217.0 /24.

1.1.4 Application Details

The additional software utilized:

- Logstash 7.6.0: The following modules are used to structure the log flow as it enters and leaves the Logstash service.
    - TCP Input
    - UDP Output
    - UDP Input
    - File Output
    - File Input
- Python 3.6 & Modules / Libraries: These modules allow for the interaction between the Operating System (OS) and the log files. These specific modules allow for the structuring of timestamps, filenames, and file/string hashes that are used throughout the network.
    - Hashlib - To provide md5 checksum calculations
    - OS - file system interaction
    - PathLib - OS file statistics
    - Subprocess - Bash command interaction
    - Datetime - date / time manipulation

1.1.5 Scripting and Evaluation

The logs are processed as one message per datagram and have a maximum size of 65535 octets, not including the IP and UDP headers (Ormianski. 2009). The following inventory is used to evaluate the logs:

- MD5 hashing function determines the uniqueness of the files. If two independent systems can calculate an identical hash sum of a file, it is mathematically probable that those two files are identical.
- The diff command is used to compare two text files in a line by line manner.
- The sort command is used to ensure that log containers are structured consistently and is something that is independently reproducible.

- The host-based agent manages and processes the generated log files and is scripted with Python 3.6. The host-based agent creates and writes the data structure and packages the contents in a log file on System A. Additionally, on System B, the host-based agent verifies the received log containers.
- Splunk Search Processing Language (SPL) indicates any log disparity and visualizes the log container status.

2. Event and Log Management

At the heart of any system set up for continuous monitoring is the log plumbing. There is a vast landscape of open source and commercial tools that control logs' functionality at various levels: Splunk, Elastic, AlienVault, Rsyslog, Syslog-ng, Apache Kafka, and many others. The low flow generally supported by this paper and these products is summarized below. (Dahlqvist. 2018).

Data > Log Input > Filters > Outputs > Analysis

Storage retention and securing transmissions between hosts are both fundamental security considerations within a logging infrastructure (NIST, 2020). A guide on how to structure and present that log data is described in a National Institute of Standards and Technology publication (NIST, 2006). The separation of infrastructure, network resources, and the storage of logs within this paper only serve as a model for monitoring logs' transmission through one-way network communications.

2.1 Logstash

Logstash is a component within the Elastic stack, previously known as ELK (Elastic, Logstash, and Kibana). The ELK stack has outgrown its three-letter acronym and contains much more than the three original programs (Elastic, 2020). Logstash defines itself as a tool that can collect, process, and forward events and log messages (Elastic.co, 2018). Logstash does not perform a unique function in the context of this paper and can be replaced by any of the products previously mentioned.

Logstash is configured with a YAML format configuration file. The program is configured to load in all configuration files located in the conf.d directory. However, for this paper, only one configuration file is used on each system is located at:

```
System A : /etc/logstash/conf.d/loglossA.conf
System B : /etc/logstash/conf.d/loglossB.conf
```

Logstash requires very little in terms of system dependencies; however, two configuration items require consideration and are discussed below.

The first configuration change is to pass ownership of the directory where the logs will be written. The Logstash user and group are configured by default on the installation of Logstash.

```
#> chown -R logstash:logstash /var/log/logloss
```

The second configuration is specific to the OS and the security profiles that are in place. With CentOS 8, system ports require root privileges to set up a listener. Typically port 514 is used to pass Syslog traffic; however, to avoid running Logstash as root or introducing a port forward with the firewall, port 1514 for Syslog traffic and port 44444 for the beacon traffic were selected for this reason (IANA, 2020)

The Logstash configuration files for System A and B can be found in Appendix A and B, respectively. Logstash controls all system outputs for all log messages, whether related to network transmissions or storing logs to files.

2.1.1 Controlling Output

The two primary outputs used are UDP and File. Both outputs are native modules within the Logstash program and interface with the underlying OS without additional dependencies (Elastic.co, 2018).

System A Outputs

UDP Port 1514 - This is the primary one-way communication channel for all Syslog messages being aggregated by System A and sent to System B. The outgoing message is being sent as a JSON structure log message so that when it is received by System B, the appropriate extraction of the message can be retained.

UDP Port 44444 - This is the channel used to send the canary beacon to System B. The separation of this channel from port 1514 makes it easier to filter System B information. This separation eliminates the need for a decision tree on which format to apply to the logs. The other reason is that the log loss process can be separated between a loss in the canary beacon and a loss of the log files.

Local File Output (Log Files) - All logs entering the Logstash process must be written to file. This output allows the logs to be placed into a container, where the file processing can occur. The host-based agent processes the containers and can be visualized as sitting between the input and output configurations of Logstash. The block diagram in figure 3 (Section 3.2) displays this relationship.

System B Outputs

Local File Output (Log Files) - All system logs brought into System B are immediately transferred to file containers. The specification of the file container is identical to the specification used on System A.

Local File Output (Canary Beacon) - All beacon logs are written to file. These logs are identified and isolated and do not show up in the system log files.

2.1.2 Controlling Input

System A Inputs

TCP Port 3333 - This port is the listening port used by System A to aggregate logs from hosts to maintain two-way communications. No codecs are applied, and no assumptions are made about the data format entering this channel. In this paper, the output is generated from *.info on the local Rsyslog daemon to have a more verbose channel of messages.

Local File Input (Canary Beacon) - This input controls the UDP transmission of the canary beacon. The host-based agent processes and writes the file. The file listener detects the change and transmits the corresponding log message to System B.

System B Inputs

UDP Port 1514 - This is a listener set to receive all log messages coming from System A.

UDP Port 44444 - This is a listener set to receive the canary beacon coming from System A. These messages are tagged so when they are shipped to the output module, the canary beacon can be directed to its output file.

2.1.3 Filters

A key component to the data structure is both sides agreeing on a file and log message format since the two systems cannot communicate. Codecs are used within Logstash to force these formats along each of the logging pipelines, but when it comes time to write the log messages, any remaining codec must be overwritten, and the desired log message must be used to write to file. The output on both System A and System B is set to utilize a JSON extraction to pull the necessary JSON elements and repackage that information into a single line with the timestamp and message elements only, as shown below.

```
codec => line { format => "%{+YYYY-MM-dd-HH}:%{message}"}
```

This codec can construct a message based on the incoming JSON line shown below. A timestamp is created based on the @timestamp field, and the message value is extracted as it was initially received by System A. The rest of the values are thrown out on the final write, but they are available for logic and additional filtering purposes until they are sent to the output module.

```
{
      "@timestamp":"2020-10-06T01:36:26.575Z",
      "Host":"192.168.217.128",
      "Port":56538,
      "message":"<30>Oct  6 01:36:26 localhost systemd[1]: Starting dnf
            makecache...","@version":"1"
}
```

The output of the filter will be the message that is written to the log container.

```
2020-10-06-01:<30>Oct  6 01:36:29 localhost systemd[1]: Started dnf
makecache.
```

2.2 Splunk

Splunk will be installed on System B and acts as the collector of log files from System A. Splunk allows for the comparison of the incoming log files, the canary alert file as well as the internal

alert file. Combining these 3 data sources under a single tool makes the processing and visualization of the state of the data obvious.

The input stanzas for the alert files are:

```
[monitor:///var/log/logloss/alert.log]
disabled = false
host = reporter
sourcetype = syslog

[monitor:///var/log/logloss/alertfile_b]
disabled = false
host = turtle
sourcetype = syslog
```

Punctuation in the context of Splunk is a way to characterize the data pattern of the log file. If the punctuation is reliable, it allows for easy and consistent parsing of the log message fields. In the example below, there are four pairs of square brackets, each with three commas.

```
punct="_:--_::.:[,,,]:[,,,]:[,,,]:[,,,]"
```

Punctuation allows us to define what each element within the string will be. This process is known as data extraction. If well defined, the string between the second comma in the second set of square brackets can have an exact value. These can be thought of as a string template.

Jason Leverton, Leverton.jd@gmail.com

Type		Field	Value	Actior
Selected	✓	host ▾	reporter	
	✓	punct ▾	_:--_:::[,,,]:[,,,]:[,,,]:[,,,]	
	✓	source ▾	/var/log/logloss/alert.log	
	✓	sourcetype ▾	syslog	
Event		LineAvg ▾	193	
		Name ▾	Log Report	
		dtg2 ▾	15	
		dtg3 ▾	01.457861	
		field2 ▾	2020-09-27 01	
		hash ▾	1358d6b64975bf8442b9063666b79081	
		hour ▾	00	
		hour1 ▾	00[1358d6b64975bf8442b9063666b79081,88373,457,193]	
		hour2 ▾	23[43886185594162087d64024fc8c9100c,88581,475,186]	
		hour3 ▾	22[a21eaf24230abe8945d83938e6f7949b,87485,471,185]	
		hour4 ▾	21[ec7f9a6c63c10ac4735d50c5b1599275,87409,469,186]	
		lines ▾	457	
		size ▾	88373	

Figure 2: Splunk Data Extraction

Figure 2 shows that the data extractions for hash, size, lines, and LineAvg are defined and the ability to look at the properties for hour1, hour2, hour(n). This data allows for easy comparison between two alert files from both System A and B and allows for complex comparisons with relatively straight forward syntax. For example, the value of hour1 from alert.log and the value of hour1 from alertfile_b can be automatically extracted and evaluated.

3. Communications

This section describes the communication flow between the systems and the necessary link and IP layer communication configurations that are required.

3.1 One-way communications

One-way communications are more than just considering the transport layer protocol and making a decision about UDP or TCP. There are constraints with the underlying protocols. Certain accommodations have to be made to ensure the functionality of the log flow is

achieved. Specifically, the ARP protocol requires an initial communication to populate a table that maps IP addresses to physical addresses (Plummer, 1982). Due to this, a persistent MAC address to IP address is needed. The following changes must be issued to make this mapping and to have it be persistent:

```
vi /etc/NetworkManager/dispatcher.d/ifup-local

!/bin/bash
arp -s 172.16.0.2 76:45:aa:31:39:44
```

3.2 Logging Pipeline

With the systems and communications defined, the block diagram of the communication channels, processes, and tools can be constructed when it is all put together.

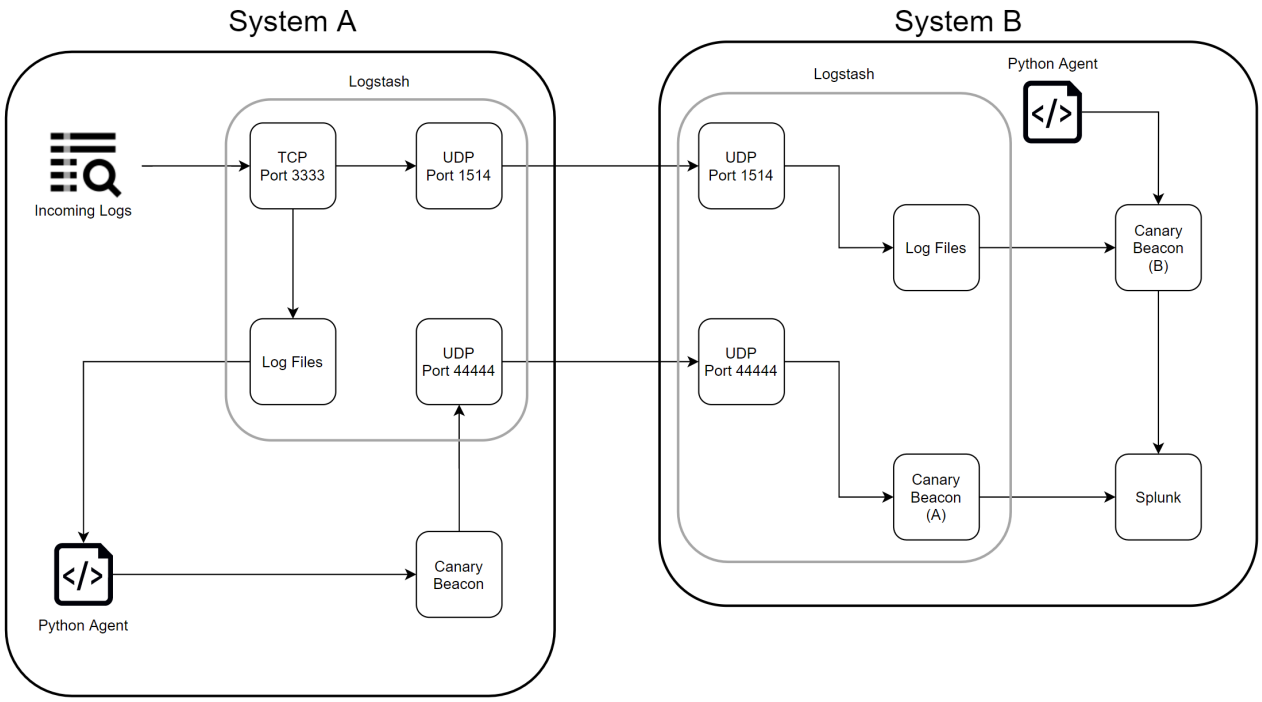

Figure 3: System-Level Block Diagram

Jason Leverton, Leverton.jd@gmail.com

4. Detection

4.1 Host-Based Agent

Like symmetric key cryptography, two endpoints can communicate if they share that same secret key. If the key were to be different from one another, the message would be lost entirely, and only random, useless data would be received.

A comparison can be drawn between this and data structures. If System A and System B are both in agreement on a pre-defined and established data structure beforehand, both sides should be able to reach a consensus on what that data looks like, or at the very least, what it should have looked like.

Since the sender can only send the receiver log messages, a secondary channel can be established to send messages about the log messages. If both the sender and receiver can perform identical tasks on a defined set of logs, the results should be identical if all logs were received in that given period. If the results are different, a second alert can be generated notifying log loss and asserting the volume, time, and duration of the log loss.

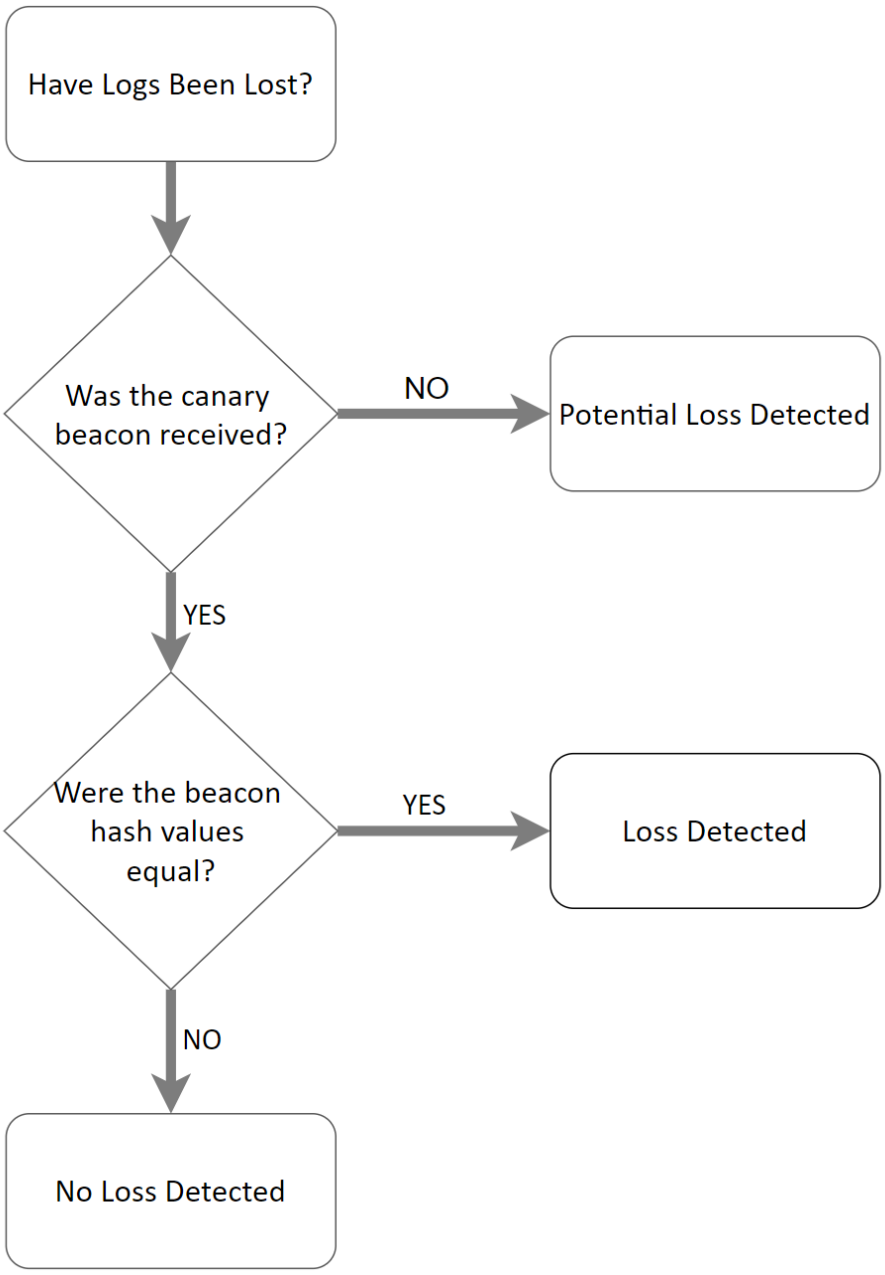

Figure 4: Data structure Log Loss Flowchart

A simplified decision flow is shown in figure 4, showing the two cases. This decision can be made on System B if the same process was used on System A. Figure 5 shows that if a container of logs can be isolated on a known time interval (t) and a unique, reproducible function (fx) can be run. The state of that log container can now be reported as loss or no-loss.

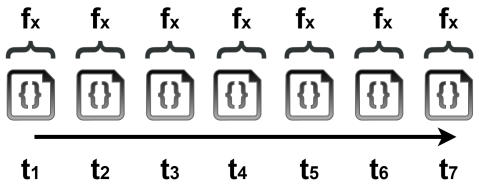

Figure 5: Time Sequenced File Functions

4.1.1 Sender Alert

The sender has an added role in this relationship of manufacturing the metadata on the log containers. In the Logstash section on the Output module, a separate communication channel is established on port 44444. This port sends the metadata exclusively. This communication has two purposes. It allows for a separation of metadata traffic from normal log flow, allowing for cleaner filtering of data. It also allows for deviations in the configuration file without altering any possible log tagging being done. The exact structure of this alert is defined in section 4.2.

4.1.2 Receiver Alert

The receiver must replicate the same processes and on the same containers to validate the log container. If the receiver can replicate the hash and the container's property values, it can accurately predict whether logs were lost in transmission for that container.

UDP does not promise that packets will be received in order, which leads to limitations and unreliable delivery. It goes as far as to say that if UDP is used, the applications must reorder packets at the application layer (Eggert, 2017). This feature was an initial setback, as the file containers on System A and System B were identical in content. However, during transmission, entries were on different line numbers. This small change meant that both the diff content and the hash values would now show non-uniqueness, or in terms of the autonomous system of detecting log loss, it was shown that log loss had occurred when it had not. Figure 6 shows how some of the log messages were received. The source file on the left and the received file on the right shows the same messages, but just in a different order.

Jason Leverton, Leverton.jd@gmail.com

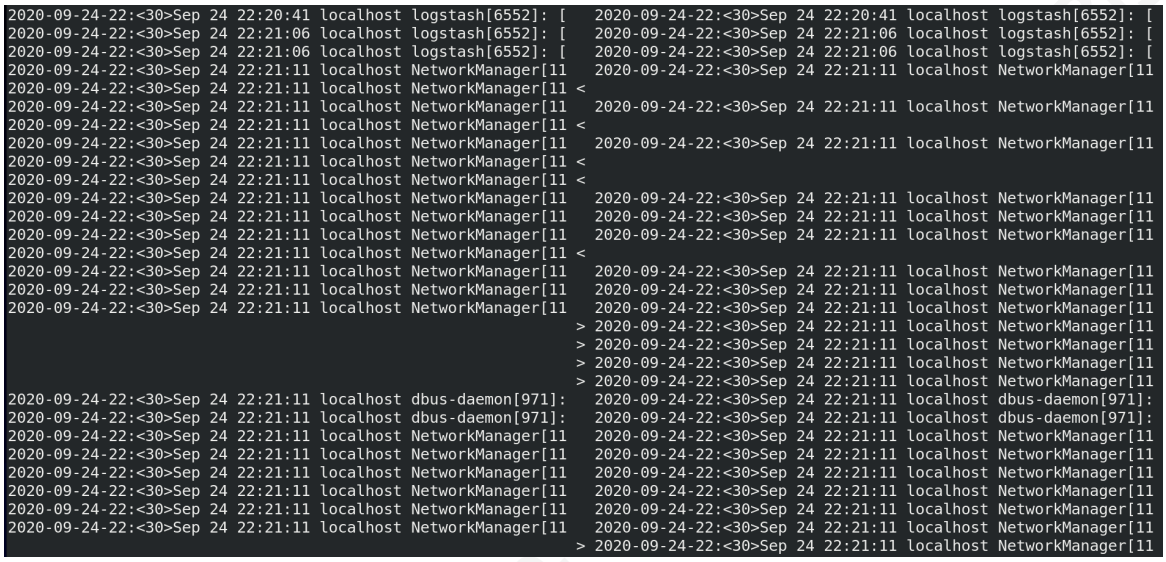

Figure 6: Diff output showing unordered log disparity between System A and System B

A sort function was inserted into the file processing flow before a hash was computed. The following commands show that once the files are sorted, the diff output below is no longer present (-q), and the md5 hash values are now equal.

```
#Testing the functionality on bash
[root@localhost]# sort sortedA -o sortedA
[root@localhost]# sort sortedB -o sortedB
[root@localhost]# diff -q sortedA sortedB
[root@localhost]# md5sum sortedA
b0db75517ef3f5c019d595c709c9df3f  sortedA
[root@localhost]# md5sum sortedB
b0db75517ef3f5c019d595c709c9df3f  sortedB
```

It should be noted that in this case, the order of the lines within the file makes no difference. Since each line is treated as a unique event, and the timestamp within each event is extracted to create a proper timeline within a security information and event management tool (SIEM). This may not always be the case if allowing multi-line events or objects. This can be solved by applying an appropriate codec within Logstash on those events' input and filtering or mutating as needed (Elastic Docs, 2020).

4.2 Data Structure and Alerting

System A and System B each play a role in how each system processes the data. How the data is packed and displayed allows both systems to communicate in a very specific way. In section 1.1.5, while discussing Splunk, the idea of event punctuation was explored. Punctuation is the result of a data structure, and if done correctly, the punctuation should have a one to one relationship with the format of the data structure. The punctuation can also be characterized as the data structure to pass information between System A and System B (Splunk Docs, 2020). Punctuation is a specific Splunk term, but it can also be decoupled and expressed more generically as a regex pattern set up to do non-greedy matches of only special characters.

Each event is packaged with the previous four hours of events. This serves several purposes. If one of these events is lost in transmission, the information can still be extracted in a follow-on message. It also aids in data integrity. Once a log container has been processed, it should become immutable. If the hash changes over time, it can demonstrate log tampering or a misconfigured process. It also provides context to any inconsistencies with the logs.

A perfect example of this is during the initial configuration of this system. Log loss errors were received between System A and System B. The hash values were different on both files, but the file size, line count, and line average showed no differences. Upon inspection, this was due to an ordering error as logs were being transmitted between the systems—the additional context allowed for quick identification of the potential issues.

The data structure of these events are:

Container Prefix : Timestamp : [hour -1]:[hour -2]:[hour -3]:[hour -4]

Inside each hour is a comma delimiter: hash, size of the file in bytes, line count of the container, and the average length of each line in the container.

The definitions that can be derived from the data structure and their placement using a colon delimiter are:

- Prefix: :[0]
- Hour -1 :[4]
- Hour -2 :[5]
- Hour -3 :[6]
- Hour -4 :[7]

The delimiter selection was arbitrary, and in hindsight, a less common delimiter could have been selected as there was an overlap with the colons from the timestamp. The colon is still functional but requires the data structure sequence to be adjusted when factoring in the colon's delimiters. This is why it jumps from 0 to 4.

The hours can be extracted within the encapsulated square quotes and delimited with the comma and defined as follows:

- Hash,[0]
- Size,[1]
- Line Count,[2]
- Average Line Length,[3]

The following log sample shows a fully populated event.

```
Log Report:
2020-09-2714:15:02.005595
:13[7272233af07779662cb45ba4e81dae56,86347,464,186]
:12[e65561317e6c31319f8870633e9b2e9f,86979,469,185]
:11[a04515c126390b67d98cd597bbdc3840,85128,453,187]
:10[0747a9833b4e1f7106a8740e80d2712b,86909,467,186]
```

5. Analysis

There are many mechanisms to introduce log loss into a system that only communicates in one direction. Since UDP is being used in this instance, there are no return messages that need to be captured or gratuitously accepted. As previously discussed, the ARP protocol does require two-way communication, but this was solved by adding a static arp entry into System A.

Since physical access is available to both systems, the files are collected on a separate data channel. This out of band channel is used to determine the exact logs that were lost in transmission. The files are now sorted text files, where each event is a string and terminated by the end of line character. The diff command highlights the exact differences between the two log containers.

The process for comparison will be as follows:

1. System A continuously deploy logs to System B using UDP in real-time.
2. System A sends a log transaction; it creates a copy and forwards that to a log container defined by a time duration of 1 hour. Specifically XX:00:00 to XX:59:999
3. System B receives a log transaction; it stores the log transaction into a log container defined by a time duration of 1 hour based on the log file's timestamp.
4. At X:15 each hour, the previous 4 log containers are all processed for the hash, size, length, and average transaction characters.
5. Once processed, the information is packaged into a data structure and shipped as a log transaction
6. System B receives the data structure and compares the information contained against its file processing to determine if log loss has occurred or not.
7. Splunk visualizes the comparison and intuits the time region and the volume of log loss.
8. The diff command outputs are used to verify the accuracy and precision of the Splunk output.

The resulting output of the process is to determine the precision and accuracy of these questions:

1. Was the process able to determine if logs were lost between System A and B?
2. Was the process able to determine the time frame of the lost logs between System A and B?
3. Was the process able to determine the volume of logs that were lost between System A and B?

5.1 Introducing Log Loss

There are two elements for introducing log loss into the system. The first involves the loss of the stream of Syslog events, and the second is the loss of the canary alert file.

For both scenarios, the firewall on System B is configured to block the specific traffic. The two open ports coming into System B are 1514 and 44444.

```
#Commands to block packets from arriving into SystemB
firewall-cmd --zone=public --remove-port=1514/udp
firewall-cmd --zone=public --remove-port=44444/udp
firewall-cmd --runtime-to-permanent
firewall-cmd --reload
```

```
#Commands to allow packets into System B
firewall-cmd --zone=public --permanent --add-port=1514/udp
firewall-cmd --zone=public --permanent --add-port=44444/udp
firewall-cmd --reload
```

5.2 Diff Output and Controlling Loss

The Diff command is an excellent tool for this situation. Since the logs are structured to be one event per line, and since both systems can be accessed through alternate data paths, this allows the exact extraction of the lost logs. Since the outages are controlled, the containers can be immediately isolated. In this case, it is the container from 17:00:00.000-17:59:59.999.

The output from the diff command is shown below. The command is set only to show the lines that are different between the two files. The line count option is used at the end (wc -l), which provides the number of lost logs within this container. Diff shows that 28 events between 17:03:45 and 17:07:46 were not received by System B, and there is a byte difference of 5830 bytes. (Linux Manual, 2020)

```
[root@localhost diffOutput]# diff --suppress-common-lines -y 17fromA
17fromB
2020-09-26-17:<12>Sep 26 17:03:45 localhost org.gnome.Shell.d <
2020-09-26-17:<12>Sep 26 17:03:45 localhost org.gnome.Shell.d <
2020-09-26-17:<30>Sep 26 17:02:46 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:02:46 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:03:06 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:03:06 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:03:31 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:03:31 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:04:11 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:04:11 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:04:36 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:04:36 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:04:56 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:04:56 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:05:21 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:05:21 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:05:46 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:05:46 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:06:16 localhost logstash[7755]: [ <
```

```
2020-09-26-17:<30>Sep 26 17:06:16 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:06:41 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:06:41 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:07:06 localhost logstash[7755]: [ <
2020-09-26-17:<30>Sep 26 17:07:06 localhost logstash[7755]: [ <
2020-09-26-17:<37>Sep 26 17:03:38 localhost su[27512]: (to ro <
2020-09-26-17:<86>Sep 26 17:03:38 localhost su[27512]: pam_un <
2020-09-26-17:<86>Sep 26 17:03:38 localhost su[27512]: pam_un <
2020-09-26-17:<87>Sep 26 17:03:38 localhost su[27512]: pam_sy <
[root@localhost diffOutput]# diff --suppress-common-lines -y 17fromA
17fromB | wc -l
28
[root@localhost diffOutput]# ls -l
total 172
-rw-r--r-- 1 logstash logstash 89319 Sep 26 21:15 17fromA
-rw-r--r-- 1 alab     alab     83489 Sep 27 04:38 17fromB
[root@localhost diffOutput]# expr 89319 - 83489
5830
```

On the System B side, during this same period, it can be seen that Splunk has detected that the log containers are showing an alert. In figure 7, the boxes represent the hourly containers and the applicable processing of the hash values within the canary beacon from system A and the local processing of System B. This specific alert is looking for two hash values from two separate log files to be equal. If two identical hashes are present, the system recognizes this as useful and highlights the value as green. When different hashes are detected, this signals a problem with that container.

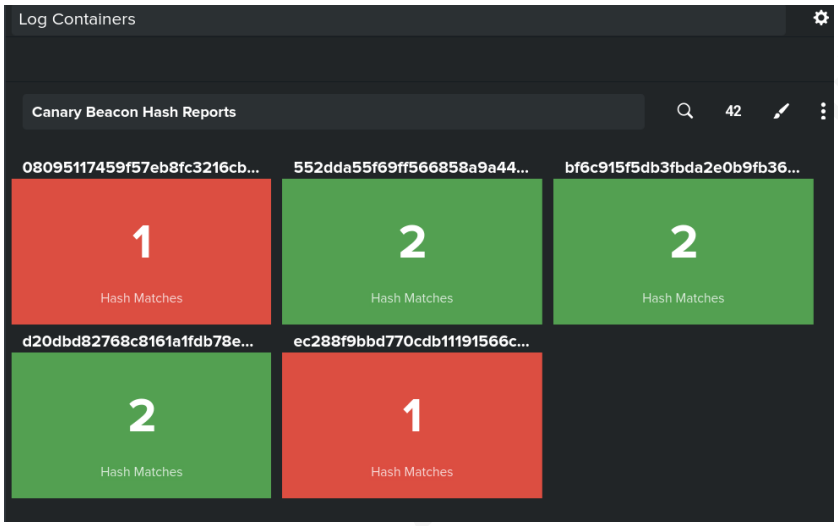

Figure 7: Detection of Hash Errors in Splunk.

The resulting log files show the actual event data that is driving this visual for the hour 17 log container.

9/26/20 6:15:01.500 PM
Log Report:2020-09-26 18:15:01.541345:17[08095117459f57eb8fc3216cbf7e0aa4,83489,451,185]:16[d20dbd82768c8161a1fdb78e517f0079,88077,475,185]:15[bf6c915f5db3fbda2e0b9fb36b1baeca,89137,477,186]:14[552dda55f69ff566858a9a446aa51fa4,89181,477,186]
host = turtle | source = /var/log/logloss/alertfile_b | sourcetype = syslog

9/26/20 6:15:01.500 PM
Log Report:2020-09-26 18:15:01.501967:17[ec288f9bbd770cdb11191566c5cc6f60,89319,479,186]:16[d20dbd82768c8161a1fdb78e517f0079,88077,475,185]:15[bf6c915f5db3fbda2e0b9fb36b1baeca,89137,477,186]:14[552dda55f69ff566858a9a446aa51fa4,89181,477,186]
host = reporter | source = /var/log/logloss/alert.log | sourcetype = syslog

Figure 8: Raw Event Output of Log Loss

The raw events also show that System B processed (top event) values of a file size of 83489 bytes received; the file has 451 lines, and the average line length was 185 characters. System A reported (bottom event) that it had sent 89319 bytes, 479 lines and that the average line length was 186 characters.

	Calculated	Out of Band

	Processing	Processing (exact)
Detected Log Loss	Yes	Yes
Difference in Size	5830 bytes	5830 bytes
Difference in Line Count	28 lines	28 lines
Time of Loss	17:00:00.000-17:59:59.999	17:03:45 - 17:07:46

Table 1: Comparison of Results Between Processed and Controlled

The results demonstrate that successful detection was made and that the quantity and volume were reported precisely. These results provide a sense of the scale of the loss, the types of logs lost. The time range within the container remains unknown. However, this allows an incident to be initiated to reconcile the two long containers' state. This stage would depend highly on the procedures of the local SOC element responsible for the logs (Taylor, 2016).

The problem of the time fidelity is amplified if a log error crosses the hour boundary. If a log loss event occurs from 17:58 to 18:02, the tool reports two failed log containers and a log loss event from 17:00 to 18:59. Similarly, two separate log events within the hour only show a single event, as the errors were all aggregated within the container size boundary.

The next stage of loss addresses the dependency of the canary beacon file. Without this event, only one side of the story can be put together, and is impossible to determine if log loss has occurred. There is an amount of robustness built into the logging framework. If a single event is missed, the next beacons contain the hash values for the missed hour. There are three follow-on periods for a four-hour window that the beacon could reconcile the missing hour. This depends on the SOC procedures, but an event can be initiated on a single lost report at the expected interval.

If the canary beacon is lost in transmission, the offending alert on System B drives this activity. If this beacon is not received between the XX:14-XX:16, it immediately is flagged as missing on System B and initiates a separate alert. This type of alert would be something a SOC would initiate according to their procedures.

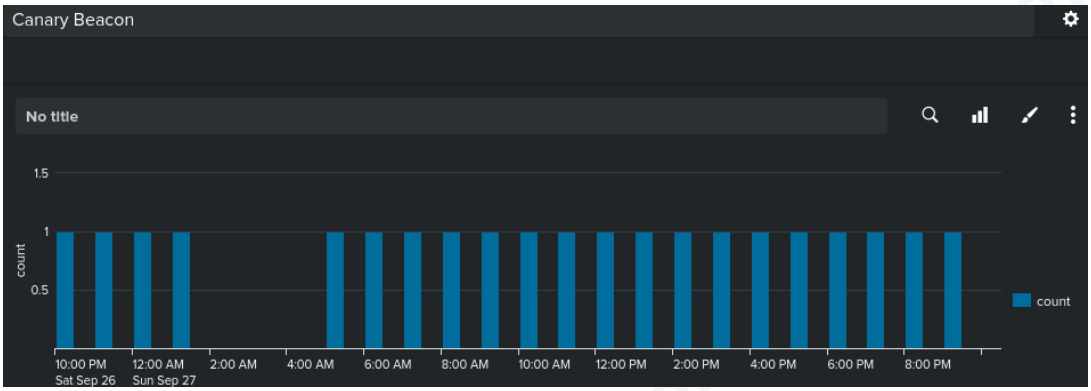

Figure 9: Canary Beacon Missed Window

Figure 9 shows the visualization when a log message is not delivered. These periodic reporting intervals make it easy to trigger an incident based on a missed log. Without the canary beacon, all log containers report as compromised, so the entire system depends on the delivery of these packets. This loss could become problematic for incident response teams if the system is poorly designed and intermittent loss is regularly occurring. The network resources that were used throughout this paper were utilized to ensure that the underlying network implementation was as reliable as it could be, so a very high delivery rate on UDP messages could be achieved (Okmianski, 2009).

Figure 10 shows the expected behavior over a four-hour window where all containers are reporting consistent hash values, and the beacons are all received over a four-hour window.

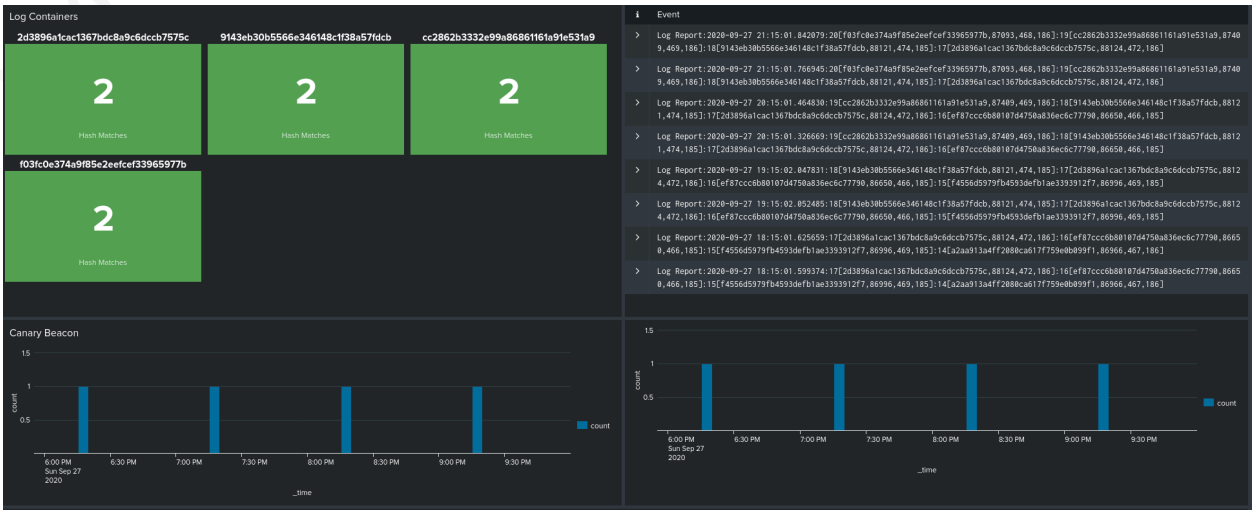

Figure 10: Splunk Dashboard Overlay

6. Conclusion

While UDP is a fundamentally poor choice to ship and move logs between networks, there are times when its use is unavoidable, such as compatibility, restrictions on trust boundaries, or consequences of bad design. There are tools available that can mitigate some of the concerns inherent with UDP.

This paper aimed to show that log loss could be detected on UDP aggregators, even if the communication was one-way. This detection was done by creating log containers defined by periodic intervals. While this paper used one-hour time containers, any time interval could be employed. Once a time interval was reached, the log container would have a local host-based agent on both sides of the UDP channel to perform identical operations on the log container. These operations would extract the metadata on the file and compare the two independent processes to determine if the log containers were identical or any loss detected; the loss could be categorized by the loss of bytes and the loss of events. The limitation of using log containers with time intervals meant that the log loss window's resolution would be equal to the log container's size. In the case of a one-hour container, the resolution of log loss would be one-hour.

Three questions were initially posed as the goal of detection:

Was the process able to determine if logs were lost between System A and B?

> The use of a defined logging standard and file hashing meant that log loss was successfully and reliably detected. File hashing was made possible with the reliable processing of the data with a host-based agent on both System A and System B. A defined logging standard meant that both systems could structure their log containers consistently.

Was the process able to determine the timeframe of the lost logs between System A and B?

> The resolution of lost logs is proportionate to the size of the log container value. There is no reason the container value could not be reduced to minimize the window entirely. The limiting factor on this as the window became much smaller, is that it was noticed that some logs only maintained a resolution of +/- 1 second on the log timestamps. There would be some rounding issues and would result in some logs rounding down for the con.

Was the process able to determine the volume of logs that were lost between System A and B?

The file size was reliably reported both as a function of a line count and byte size of the log container. The nature of the host-based agents running on both systems was consistent and reliable.

The final result was that log loss was able to be reliably detected. The introduction of the canary beacon and the follow-on action to its non-delivery was an integral element to the detection. If a known message and interval can be established at two ends, an event can be negotiated on the delivery and non-delivery of that message.

7. References

Dahlqvist. (2018). A Practical Introduction to Logstash: https://www.elastic.co/blog/a-practical-introduction-to-logstash

Eggert. (2017). RFC 826 - UDP Usage Guidelines https://tools.ietf.org/html/rfc8085

Elastic.co. (2018). Installing Logstash https://www.elastic.co/guide/en/logstash/current/installing-logstash.html

Elastic.co. (2020). What is the ELK Stack? https://www.elastic.co/what-is/elk-stack

Elastic Docs. (2020). Managing Multi-line Events https://www.elastic.co/guide/en/logstash/current/multiline.html

IANA. (2020). Service Name and Transport Protocol Port Number Registry https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt

Ger. (2017). Small, Medium, or Large - Scaling Elasticsearch and Evolving the Elastic Stack to Fit: https://www.elastic.co/blog/small-medium-or-large-scaling-elasticsearch-and-evolving-the-elastic-stack-to-fit

Gerhards. (2009). The Syslog Protocol: https://tools.ietf.org/html/rfc5424

Gerhards. (2008). On the (uGern)reliability of plain tcp Syslog… https://rainer.gerhards.net/2008/04/on-unreliability-of-plain-tcp-syslog.html

Linux Manual. (2020). diff(1) — Linux manual page: https://man7.org/linux/man-pages/man1/diff.1.html

NIST. (2020). Control Baselines for Information Systems and Organizations https://csrc.nist.gov/publications/detail/sp/800-53b/draft

NIST (2006). (2006). Guide to Computer Security Log Management: https://csrc.nist.rip/library/NIST%20SP%20800-092%20Guide%20to%20Computer%20Security%20Log%20Management,%202006-09.pdf

Okmianski. (2009). Transmission of Syslog Messages over UDP:

https://tools.ietf.org/html/rfc5426

Plummer. (1982). Address Resolution Protocol:
https://tools.ietf.org/html/rfc826

SANS ICS Blog. (2019). The Risks of an IT Versus OT Paradigm:
https://www.sans.org/blog/the-risks-of-an-it-versus-ot-paradigm/

SANS Newsletter. (2020). Hackers are Using Malware Designed to Target Airgapped Networks
https://www.sans.org/newsletters/newsbites/xxii/40

Splunk Docs. (2020). Splexicon: Punct
https://docs.splunk.com/Splexicon:Punct

Syslog-ng. (2018). Collecting log messages from UDP sources:
https://www.syslog-ng.com/technical-documents/doc/syslog-ng-premium-edition/7.0.9/collecting-log-messages-from-udp-sources

Syslog-ng. (2019). Improved log collection over UDP:
https://www.syslog-ng.com/community/b/blog/posts/improved-log-collection-over-udp

Taylor. (2016). Continuous Monitoring: Build A World-Class Monitoring System for Enterprise, Small Office, or Home:
https://www.sans.org/reading-room/whitepapers/detection/continuous-monitoring-build-world-class-monitoring-system-enterprise-small-office-home-37477

Todd (2017). Creating a Logging Infrastructure:
https://www.sans.org/reading-room/whitepapers/logging/creating-logging-infrastructure-38130

Chukwa (2010). Chukwa: A system for reliable large-scale log collection:
https://www.usenix.org/legacy/events/lisa10/tech/full_papers/lisa10_proceedings.pdf#page=171

8 Appendix A - Logstash Configuration System A

```
input {
       tcp {
              id => "input_logs"
              port => 3333
       }
}

input {
       file {
              tags => ["canary"]
              id => "alertfile"
              path=>"/var/log/logloss/alertfile"
       }
}

output {

       if "canary" in [tags] {
              udp {
                     id => "canary"
                     port => 44444
                     host => "192.168.217.129"
              }

       }

       else {
              udp {
                     id => "log_line"
                     port => 1514
                     host => "192.168.217.129"
                     codec => json
              }
       }

       file {
              id => "verify_file"
              path=>"/var/log/logloss/%{host}/incominglogs-%{+YYYY-MM-dd-HH}.log"
              codec => line { format => "%{+YYYY-MM-dd-HH}:%{message}"}
       }
}
```

9 Appendix B - Logstash Configuration System B

```
input {
       udp {
              id => "log_line"
              port => 1514
              codec => json
       }
       udp {
              id => "canary"
              port => 44444
              codec => json
              tags => ["canary"]
       }
}


output {
       if "canary" in [tags] {
              file {
                     path => "/var/log/logloss/alert.log"
                     codec => line {format => "%{message}"}
              }
       }
       else {
              file {
                     path => "/var/log/logloss/incominglogs-%{+YYYY-MM-dd-HH}.log"
                     codec => line { format => "%{+YYYY-MM-dd-HH}:%{message}"}
              }
       }
}
```

10 Appendix C - Python Script System A

```
#!/usr/bin/python3.6
import os, hashlib, socket, subprocess
from datetime import date, time, datetime
from pathlib import Path

#      Returns the average character count over the file
def AvgSizeOfLine(filename):
       lineSize = []
       with open(filename) as f:
              Lines = f.readlines()
              for line in Lines:
```

```
                        lineSize.append(int(len(line)))
        return int(sum(lineSize)/len(lineSize))

#       A function to work with the string & int properties of time.
#       Goes back in time to grab the last 4 hours, considering that the
#       pattern could be 1,0,23,22
#       Also, require leading zero for the filename, but an int property removes
#       the leading zero.  Repackage as a string and add the leading zero to both
#       hour and day.
def getPreviousFiles(currentHour, currentDay):
        a = [0,0,0,0]
        b = [0,0,0,0]
        currentHourInt = int(currentHour)
        currentDayInt = int(currentDay)

        for index in range(len(a)):
                nextHour = currentHourInt - index
                if (nextHour-1) < 0:
                        a[index]=nextHour+23
                        b[index]=currentDayInt-1
                elif nextHour == 0:
                        a[index] = 0
                        b[index]=currentDayInt
                else:
                        a[index]=nextHour-1
                        b[index]=currentDayInt

        for index in range(len(a)):
                if a[index] < 10:
                        a[index] = "0"+str(a[index])
                else:
                        a[index] = str(a[index])

                if b[index] < 10:
                        b[index] = "0"+str(b[index])
                else:
                        b[index] = str(b[index])
        return a,b

#       Returns md5 hash of the file.
def hashmd5(filename):
        filehash = hashlib.md5()
        with open(filename, "rb") as f:
                for chunk in iter(lambda: f.read(4096), b""):
                        filehash.update(chunk)
        return filehash.hexdigest()

#       Returns the amount of lines in the file.
def LineCounts(filename):
        with open(filename) as f:
                for i, l in enumerate(f):
                        pass
```

Jason Leverton, Leverton.jd@gmail.com

```
        return i+1

#       Returns a set of hash, file size, line count and avg line size as a set
#       Returns an empty set if there is no file, or the files empty.
def processFile(filename):
        if os.path.isfile(filename):
                command = "sort " + filename + " -o "+filename
                process = subprocess.Popen(command.split(), stdout=subprocess.PIPE)
                output,error = process.communicate()

                with open(filename) as f:
                        fileHash = hashmd5(filename)
                        fileSize = Path(filename).stat().st_size
                        fileLineAvg = AvgSizeOfLine(filename)
                        fileLineCount = LineCounts(filename)

                        r = [fileHash,fileSize,fileLineCount,fileLineAvg]
        else:
                print("File has no logs")
                r = [0,0,0,0]

        return r

#       Writes out to the alert file to send to System B.
def writeAlert(msg):
        alertf = open("/var/log/logloss/alertfile","a")
        alertf.write(msg)
        alertf.close()

today = datetime.now()
currentDay = today.strftime("%d")
currentHour = today.strftime("%H")
currentMonth = today.strftime("%m")
currentYear = today.strftime("%Y")

a,b = getPreviousFiles(currentHour,currentDay)

file2 = "/var/log/logloss/192.168.217.128/incominglogs-"+str(currentYear)+"-
"+str(currentMonth)+"-"+b[3]+"-"+a[0]+".log"
file3 = "/var/log/logloss/192.168.217.128/incominglogs-"+str(currentYear)+"-
"+str(currentMonth)+"-"+b[3]+"-"+a[1]+".log"
file4 = "/var/log/logloss/192.168.217.128/incominglogs-"+str(currentYear)+"-
"+str(currentMonth)+"-"+b[3]+"-"+a[2]+".log"
file5 = "/var/log/logloss/192.168.217.128/incominglogs-"+str(currentYear)+"-
"+str(currentMonth)+"-"+b[3]+"-"+a[3]+".log"

file2p = processFile(file2)
file3p = processFile(file3)
file4p = processFile(file4)
file5p = processFile(file5)

print(file2p)
```

Jason Leverton, Leverton.jd@gmail.com

```
print(file3p)
print(file4p)
print(file5p)

msg2  = a[0] + "[" + str(file2p[0]) + "," + str(file2p[1]) + "," + str(file2p[2]) + "," +
str(file2p[3]) + "]"
msg3  = a[1] + "[" + str(file3p[0]) + "," + str(file3p[1]) + "," + str(file3p[2]) + "," +
str(file3p[3]) + "]"
msg4  = a[2] + "[" + str(file4p[0]) + "," + str(file4p[1]) + "," + str(file4p[2]) + "," +
str(file4p[3]) + "]"
msg5  = a[3] + "[" + str(file5p[0]) + "," + str(file5p[1]) + "," + str(file5p[2]) + "," +
str(file5p[3]) + "]"
msg = "Log Report:"+ str(today) + ":" + msg2 + ":" + msg3 + ":" + msg4 + ":" + msg5 +"\n"

print(msg)
writeAlert(msg)
```

11 Appendix D - Python Script System B

```
#!/usr/bin/python3.6
import os, hashlib, socket, subprocess
from datetime import date, time, datetime
from pathlib import Path


#      Returns the average character count over the file
def AvgSizeOfLine(filename):
       lineSize = []
       with open(filename) as f:
              Lines = f.readlines()
              for line in Lines:
                     lineSize.append(int(len(line)))

       return int(sum(lineSize)/len(lineSize))

#      A function to work with the string & int properties of time.
#      Goes back in time to grab the last 4 hours, considering that the
#      pattern could be 1,0,23,22
#      Also, require leading zero for the filename, but an int property removes
#      the leading zero.  Repackage as a string and add the leading zero to both
#      hour and day.
def getPreviousFiles(currentHour, currentDay):
       a = [0,0,0,0]
       b = [0,0,0,0]
       currentHourInt = int(currentHour)
       currentDayInt = int(currentDay)
```

```
	for index in range(len(a)):
		nextHour = currentHourInt - index
		if (nextHour-1) < 0:
			a[index]=nextHour+23
			b[index]=currentDayInt-1
		elif nextHour == 0:
			a[index] = 0
			b[index]=currentDayInt
		else:
			a[index]=nextHour-1
			b[index]=currentDayInt

	for index in range(len(a)):

		if a[index] < 10:
			a[index] = "0"+str(a[index])
		else:
			a[index] = str(a[index])

		if b[index] < 10:
			b[index] = "0"+str(b[index])
		else:
			b[index] = str(b[index])

	return a,b

#	Returns md5 hash of the file.
def hashmd5(filename):

	filehash = hashlib.md5()
	with open(filename, "rb") as f:
		for chunk in iter(lambda: f.read(4096), b""):
			filehash.update(chunk)

	return filehash.hexdigest()

#	Returns the amount of lines in the file.
def LineCounts(filename):
	with open(filename) as f:
		for i, l in enumerate(f):
			pass

	return i+1

#	Returns a set of hash, file size, line count and avg line size as a set
#	Returns an empty set if there is no file, or the files empty.
def processFile(filename):

	if os.path.isfile(filename):

		command = "sort " + filename + " -o "+filename
		process = subprocess.Popen(command.split(), stdout=subprocess.PIPE)
```

```
                output,error = process.communicate()

                with open(filename) as f:

                        fileHash = hashmd5(filename)
                        fileSize = Path(filename).stat().st_size
                        fileLineAvg = AvgSizeOfLine(filename)
                        fileLineCount = LineCounts(filename)

                        r = [fileHash,fileSize,fileLineCount,fileLineAvg]

        else:
                print("File has no logs")
                r = [0,0,0,0]

        return r

#       Writes out to the alert file for the local processing of the logs.
def writeAlert(msg):
        alertf = open("/var/log/logloss/alertfile_b","a")
        alertf.write(msg)
        alertf.close()

today = datetime.now()
currentDay = today.strftime("%d")
currentHour = today.strftime("%H")
currentMonth = today.strftime("%m")
currentYear = today.strftime("%Y")

a,b = getPreviousFiles(currentHour,currentDay)

file2 = "/var/log/logloss/incominglogs-"+str(currentYear)+"-"+str(currentMonth)+"-"+b[3]+"-
"+a[0]+".log"
file3 = "/var/log/logloss/incominglogs-"+str(currentYear)+"-"+str(currentMonth)+"-"+b[3]+"-
"+a[1]+".log"
file4 = "/var/log/logloss/incominglogs-"+str(currentYear)+"-"+str(currentMonth)+"-"+b[3]+"-
"+a[2]+".log"
file5 = "/var/log/logloss/incominglogs-"+str(currentYear)+"-"+str(currentMonth)+"-"+b[3]+"-
"+a[3]+".log"

file2p = processFile(file2)
file3p = processFile(file3)
file4p = processFile(file4)
file5p = processFile(file5)

print(file2p)
print(file3p)
print(file4p)
print(file5p)

msg2  = a[0] + "[" + str(file2p[0]) + "," + str(file2p[1]) + "," + str(file2p[2]) + "," +
str(file2p[3]) + "]"
```

Jason Leverton, Leverton.jd@gmail.com

```
msg3  = a[1] + "[" + str(file3p[0]) + "," + str(file3p[1]) + "," + str(file3p[2]) + "," +
str(file3p[3]) + "]"
msg4  = a[2] + "[" + str(file4p[0]) + "," + str(file4p[1]) + "," + str(file4p[2]) + "," +
str(file4p[3]) + "]"
msg5  = a[3] + "[" + str(file5p[0]) + "," + str(file5p[1]) + "," + str(file5p[2]) + "," +
str(file5p[3]) + "]"
msg = "Log Report:"+ str(today) + ":" + msg2 + ":" + msg3 + ":" + msg4 + ":" + msg5 +"\n"
print(msg)
writeAlert(msg)
```