ORACLE
ZFS STORAGE APPLIANCE

Splunk and the Oracle ZFS Storage Appliance

ORACLE TECHNICAL WHITE PAPER | SEPTEMBER 2015

ORACLE

Table of Contents

Introduction 1

Overview of the Example System 2

- Why Use a Forwarder? 2
- Configuring the Oracle ZFS Storage Appliance 3

Installing the Universal Forwarder 5

Collecting Data from the Oracle ZFS Storage Appliance 6

- Configuring the Forwarders to Parse Timestamps 6

Collecting Datasets from the Oracle ZFS Storage Appliance 8

- The `get_datasets.py` Script 8
- Defining the Script 8

Using the Oracle ZFS Storage Appliance Dataset Data 10

- Example: Charting I/O Operations 12
- Example: Charting I/O Operations with Drilldown Data 15

Collecting Storage Data from the Oracle ZFS Storage Appliance 17

- The `get_storage.py` Script 17
- Defining the Script 17

Using the Oracle ZFS Storage Appliance Storage Data 18

Predictions Using the Oracle ZFS Storage Appliance Storage Data 19

Collecting Logs from the Oracle ZFS Storage Appliance 21

- The `get_logs.py` Script 21
- Defining a Monitor 22

Using the Oracle ZFS Storage Appliance Log Data 24

Example: Charting Logins 24

Conclusion 25

References 26

Appendix A: Python Code for `get_datasets.py` 27

Appendix B: Python Code for `get_storage.py` 28

Appendix C: Python Code for `get_logs.py` 30

Appendix D: Python code for `zfs_session.py` 33

Appendix E: Adding a Limited-Privilege User to the Oracle ZFS Storage Appliance 37

Appendix F: Enabling the RESTful Interface 39

Introduction

The Operational Intelligence platform Splunk Enterprise is used by many organizations to gather machine data from many sources. The collected data is then used to monitor the end-to-end infrastructure to help avoid service degradation and to troubleshoot problems such as performance bottlenecks. Splunk also provides predictive analysis tools to help determine when resources may become overburdened.

This white paper provides information on the fundamentals of collecting data from the Oracle ZFS Storage Appliance, basics on a Splunk infrastructure that allows distributing the load of collecting the data, and configuring Splunk Enterprise to index that data. It provides, through example, guidance on using this gathered data to help predict certain performance bottlenecks. This paper is intended for an audience already familiar with Splunk Enterprise, its command line interface (CLI), and the Oracle ZFS Storage Appliance.

Overview of the Example System

The Splunk Enterprise infrastructure is flexible in how it collects its machine data. The heart of the Splunk infrastructure is the Indexer node. The Indexer node transforms the raw data into events and places the result into an index. It also searches the index in response to search requests.[1]

In smaller deployments, the Indexer may also perform the other fundamental Splunk operations – data input and search management – but in larger deployments, these operations may be offloaded to other machines. A machine that takes on the data input operation is called a Forwarder. A Forwarder node runs a streamlined, dedicated version of Splunk Enterprise that contains only the essential components needed to forward data.[2]

Why Use a Forwarder?

In the Splunk infrastructure, the Indexer nodes can be quite heavily loaded, especially in a large environment. The tasks of data input can accumulate quickly, leading to poor performance at both server and network chokepoints. By using Forwarders, the machine resources can be distributed across a number of small machines or virtual machines (VMs) and the generated network traffic is more easily balanced.

In an environment where the monitored systems are spread across networks, Forwarders automatically queue the data if connection to the Indexers is lost, whereas if data is streaming directly to an Indexer, events can be lost during the network down time.

Another benefit to the use of Forwarders is the ability to group data from related machines, allowing faster access to related events on the Indexers.

[1] http://docs.splunk.com/Splexicon:Indexer
[2] http://docs.splunk.com/Splexicon:Forwarder

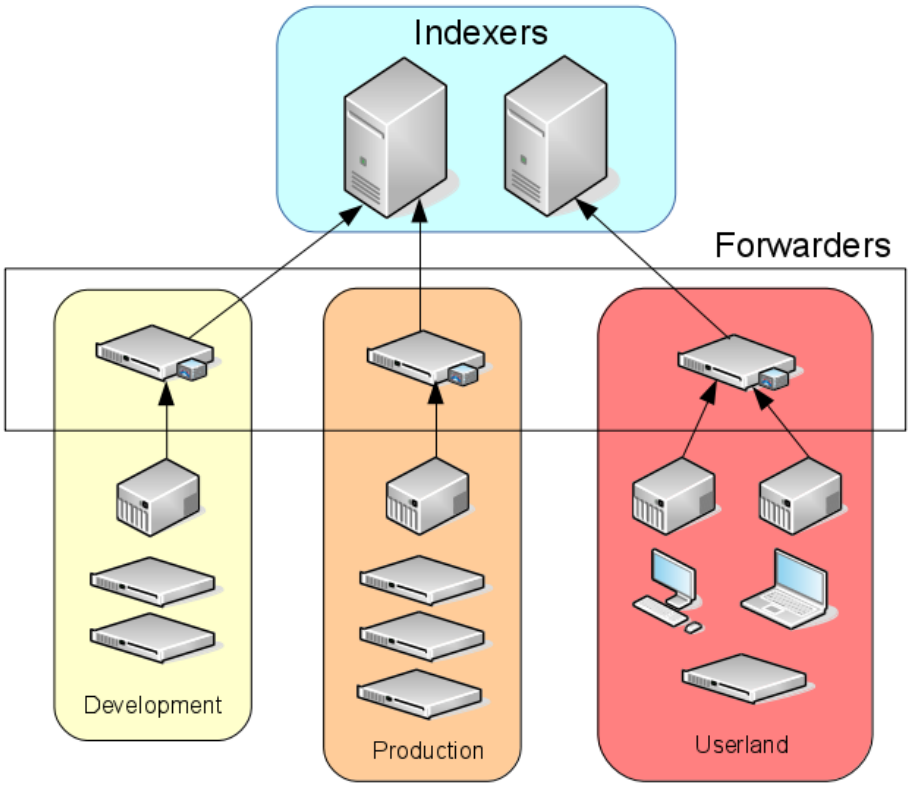

Figure 1. Distributed Splunk infrastructure

Figure 1 illustrates an example of a distributed Splunk infrastructure. It covers three distinct islands of functionality, each on a separate subnet with limited routing between them. Each island has one or more Oracle ZFS Storage Appliances containing data pulled by a Forwarder which processes the raw data and pushes it to an Indexer.

This white paper details how data from the Oracle ZFS Storage Appliance in Development can be collected by a Forwarder and pushed to an Indexer.

This example assumes an environment with an Indexer node with Splunk Enterprise 6.2 already installed and a server running Oracle Linux 7.1 with the Splunk Universal Forwarder installed. However, the concepts in this white paper are OS-independent. The Splunk Forwarder is also available for Oracle Solaris and other operating systems. As long as the Python language and the required modules are available, the implementation described here will be applicable.

Configuring the Oracle ZFS Storage Appliance

To access the Representational State Transfer (REST) – also called RESTful – interface on Oracle ZFS Storage Appliances running OS8.3 and earlier, REST must first be enabled. Please see Appendix F for details on how to enable the RESTful API. Oracle ZFS Storage Appliances running OS8.4 and above have the API enabled by default.

Configuring the Indexer

In order to accept the data from the Forwarder, the Indexer must first be configured to receive it. This is done with a Receiver process. Enable the Receiver process at the Indexer's command line interface with the command:

```
splunk enable listen 9997
```

Note that by convention, Splunk Forwarders use port 9997 to transfer data, but you can specify any unused port. If no authenticated session to the Indexer already exists, the Indexer will prompt for a username and password.

By setting up the Receiver, the Indexer will automatically add the data inputs for the files or streams it receives.

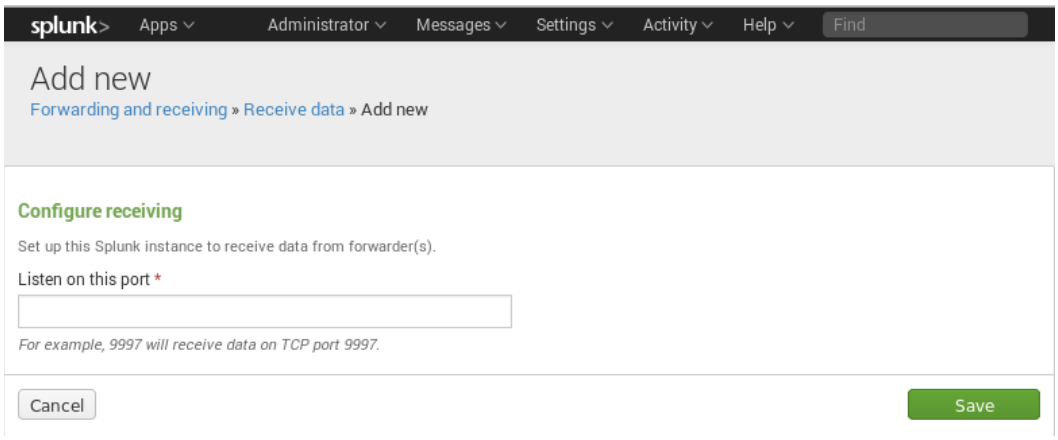

Figure 2. Adding a Receiver through the Splunk web interface

As shown in Figure 2, you can also define the Receiver from the Indexer's web GUI. Under the Settings menu, in the Data section, choose "Forwarding and receiving". Under "Receive data", click "Add new", then enter the port number of the Receiver.

Once the Receiver is in place, it will automatically ingest the data pushed to it by the Forwarder.

Installing the Universal Forwarder

The high-level steps for installation and configuration of the Linux Forwarder are:

1. If you have not already done so, download and install the Splunk Universal Forwarder on the Linux machine according to the Splunk documentation.[3] For Oracle Linux, use the RPM file for your architecture and install it with `yum` or `rpm`. The Forwarder may run as any user on the system. If it is to be run as a non-root user, ensure that the user has the appropriate permissions to read the inputs that will be specified.

2. Set the environment variable *$SPLUNK_HOME* to the directory the Forwarder was installed to. If the Forwarder is installed from an RPM package using its defaults, this will be the `/opt/splunkforwarder` directory. Append the *$PATH* variable to the location of the splunk executable, then start the Splunk Forwarder. The export lines may be added to your shell's initialization script to make them permanent: The first time Splunk is started, you must agree to its license terms. Appending "–`accept-license`" to the line that starts Splunk will accept the license without it being displayed.

   ```
   export SPLUNK_HOME=/opt/splunkforwarder

   export PATH=${PATH}:${SPLUNK_HOME}/bin

   splunk start --accept-license
   ```

3. The default Splunk admin password should be changed with the following command. If the CLI session on the Forwarder has not yet been authenticated, the screen interface will prompt you for a username and password. For a new installation, when prompted, use `admin` for the Splunk username and `changeme` for the password.

   ```
   splunk edit user admin -password <New_Splunk_Admin_Password> \ -
   role admin
   ```

4. Configure the Forwarder to point to a specific receiving Indexer. Specify the host by either IP address or name. The default port for Splunk communication is 9997, but you can easily change this at the Indexer end.

   ```
   splunk add forward-server <host>:<port>
   ```

The Splunk Forwarder package does not have a web interface. You must configure it from the command line or through configuration files.

[3] As of this writing, the documentation is available at http://docs.splunk.com/Documentation/Splunk/latest/Forwarding/Deployanixdfmanually

Collecting Data from the Oracle ZFS Storage Appliance

There are a number of ways that data can be collected from the Oracle ZFS Storage Appliance. These include the Browser User Interface (BUI), the Command Line Interface (CLI), and the Representational State Transfer (REST) interface. The REST interface is the most flexible and is used in this white paper for all interaction with the Oracle ZFS Storage Appliance. The REST interface will return the requested data in the JavaScript Object Notation (JSON) format, which is easily parsed by Splunk.

Configuring the Forwarders to Parse Timestamps

The Oracle ZFS Storage Appliance logs use a timestamp format that is not immediately recognized by Splunk. To allow Splunk to find and parse the timestamp within the log's JSON format, edit the file `$SPLUNK_HOME/etc/system/local/props.conf` to include the following stanza:

```
[host::<hostname_regex>]
TIME_PREFIX = “timestamp"
MAX_TIMESTAMP_LOOKAHEAD = 22
TIME_FORMAT = “%Y%m%dT%H:%M:%S”
TZ = UTC  # Dependent on environment
```

The *<hostname_regex>* field is a regular expression that will match the hostnames for the Oracle ZFS Storage Appliances. In the hypothetical/example development island, all of the Oracle ZFS Storage Appliances begin with `dev-zfs`, so the host line would be:

```
[host::dev-zfs*]
```

The `TIME_PREFIX` line identifies lines that contain a timestamp, and the `MAX_TIMESTAMP_LOOKAHEAD` line defines where in the line the timestamp begins. These values are correct for timestamps in the Oracle ZFS Storage Appliance JSON output.

The last line defines the format of the Oracle ZFS Storage Appliance's timestamp string. (Note that while the timestamp format shown is valid for all the examples in this paper, some sections of the REST API, such as the Problem service, use other formats.)

Note that if a Splunk infrastructure has multiple Forwarders collecting logs from Oracle ZFS Storage Appliances, this change must be made on each Forwarder. It is the Forwarders and not the Indexers that must have this timestamp stanza in place.

Depending on your environment, you may need to add a line for the time zone. Best practices recommend that systems use the Coordinated Universal Time (UTC) zone for their internal clock, with the local time zone set by the operating system. The Oracle ZFS Storage Appliance is generally configured this way, and would require the TZ line as shown previously. To test this, run the following command:

```
curl –k –u <user> https://<appliance>:215/api/service/v1/services/ntp
```

If the date line returned ends in “GMT+0000 (UTC)” then the TZ line should be used.

Any changes made to a Forwarder’s configuration files will not take effect until Splunk is restarted. To check the validity of the files without affecting the current operation, run the command `splunk btool check –debug`.

Collecting Datasets from the Oracle ZFS Storage Appliance

Splunk is machine data-driven, and the Oracle ZFS Storage Appliance has plenty for it to work with. The REST services give access to the Oracle ZFS Storage Appliance's datasets, allowing the user to collect various metrics and drilldown information for Splunk to assimilate. Full documentation on the available datasets is available in the Oracle ZFS Storage Appliance Analytics Guide.

Getting the datasets into Splunk requires a script on the Forwarder to log into the Oracle ZFS Storage Appliance, collect the data, and write it to `stdout`.

The `get_datasets.py` Script

The `get_datasets.py` script in Appendix A (and its companion module, `zfs_session.py` in Appendix D) provides an example method of collecting datasets and writing the output in the JSON format.

The usage of the `get_datasets.py` script is:

```
usage: get_datasets.py -d DATASET [DATASET ...]
                       -k KEYFILE  -u USERNAME  [-1]
                       zfs_appliances [zfs_appliances ...]
```

You can specify any dataset – and multiple datasets – listed in the Analytics Guide. The keyfile is a small JSON file that stores some information about the Oracle ZFS Storage Appliances accessed by the script. The username is the login user on all of the Oracle ZFS Storage Appliances. It is assumed that all Oracle ZFS Storage Appliances accessed by the same script have the same login credentials.

The script, when run, prompts for a password for the Oracle ZFS Storage Appliance username passed on the command line. It is highly recommended to use a non-privileged user to query the REST interface. Such a user can be created through the Oracle ZFS Storage Appliance BUI. When creating this user, the only role that should be applied is read-only access to Analytics. (Please see Appendix E for an example of adding such a user.) There are a number of methods that may be used to automate setting the password for the script. Be sure to implement a method consistent with the security practices in your environment.

Defining the Script

To have Splunk run the script and read its output, the script must first be in a location accessible to Splunk. `$SPLUNK_HOME/bin/scripts` is an ideal location for such scripts.

You must then add a script stanza to the file `$SPLUNK_HOME/etc/system/local/inputs.conf`

The stanza is of the format:

```
[script://<script_name> <script_parameters>]
interval = <seconds>
sourcetype = _json
```

It is good practice, especially when calling scripts that have additional parameters such as this, to use a wrapper script so that Splunk's parser does not accidentally misread characters that are special to it. Here is the wrapper script `get_datasets.sh`:

```
#!/bin/bash
/opt/splunkforwarder/bin/scripts/get_datasets.py \
       -d io.bytes[op] io.bytes nic.kilobytes[direction] \
       nfs3.ops[share] nfs3.ops[op] nfs4.ops[share] nfs4.ops[op] \
       -k /opt/splunkforwarder.tmp/get_datasets.key –u reporter \
       dev-zfs-1 dev-zfs-2]
```

Here is the example stanza in `inputs.conf`:

```
[script:///opt/splunkforwarder/bin/scripts/get_datasets.sh]
interval = 60
sourcetype = _json
```

Note that the *`sourcetype`* must be specified as `_json` or Splunk may improperly parse the output. Also note that the script must be specified with its full pathname, and that this will result in three slashes after the colon in the script definition line. The interval value sets the time between script executions in seconds.

When run, the script will gather the data for multiple datasets in one-second intervals since the last time the script was run. With the first invocation of the script, it will gather the data from the previous 24 hours.

If less granularity is needed for some datasets, the `-1` parameter can be passed to the script, and a single data point from the current moment will be returned for each specified dataset.

Here is an example stanza (which does not use a wrapper) to collect a single dataset every 30 seconds:

```
[script:///opt/splunkforwarder/bin/scripts/get_datasets.py \
       -d cpu.utilization[mode] \
       -k /opt/splunkforwarder/tmp/get_datasets2.key –u reporter \
       dev-zfs-1 dev-zfs-2 ]
interval = 30
sourcetype = _json
```

Note that a different keyfile is specified. When the same script is called from different stanzas, a recommended practice is to use different keyfiles to prevent contention, especially in cases when scripts are run at different intervals. This is because the last access time is also stored in the keyfile, and it is unlikely that this value can be accurately shared across scripts.

With the stanzas in place in `inputs.conf`, restart Splunk. The data collected by the Forwarder will automatically be passed to the Indexer.

Using the Oracle ZFS Storage Appliance Dataset Data

When the script begins generating data, the Forwarder will automatically push it to the Indexer. This can be seen in the “What to Search” section of the Search screen.

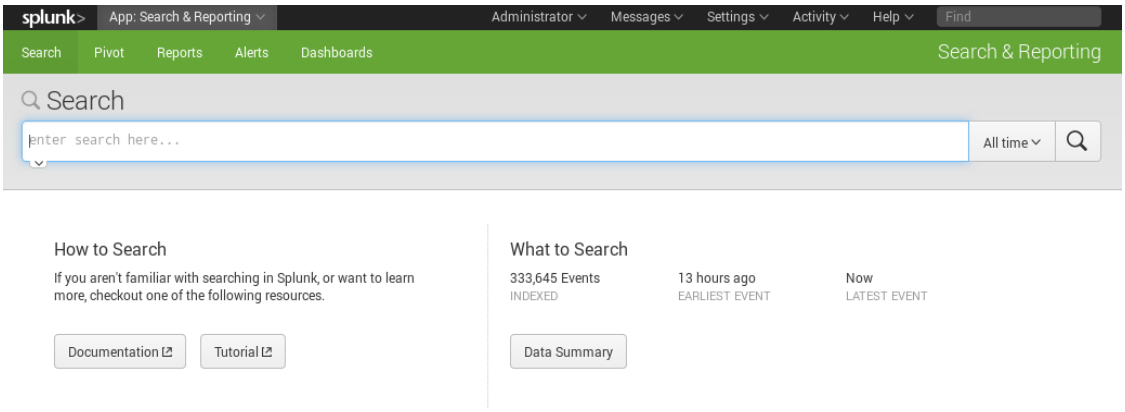

Figure 3. What to Search section in the Splunk interface

Clicking the “Data Summary” button brings up a list of hosts from which Splunk collects its data. Note that by default, Splunk uses as the “Host” the hostname of the Forwarder, not the hostname of the Oracle ZFS Storage Appliance.

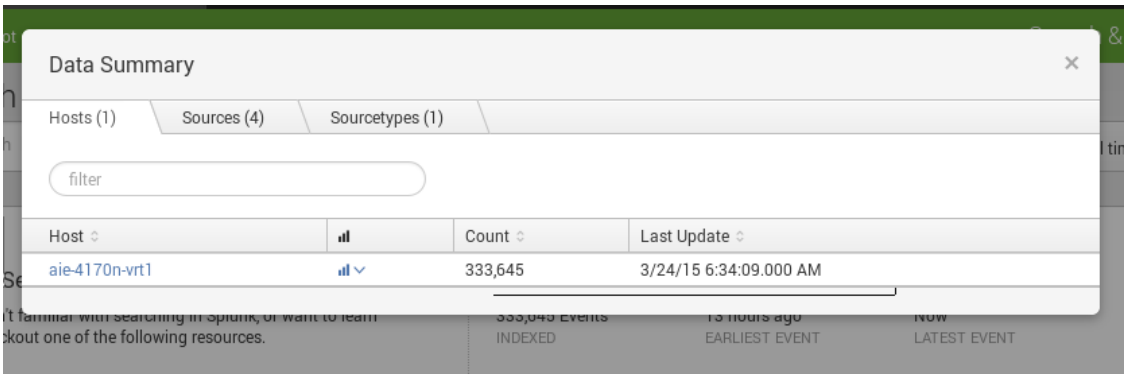

Figure 4. Data Summary

Clicking on the Forwarder’s host name will pull in all of the records that the Indexer has collected from the Forwarder. Each of the Forwarders in the Splunk infrastructure will be listed under the Host column.

In the instances where multiple Oracle ZFS Storage Appliances are reporting to a single Forwarder, a search can be created based on all records from that Forwarder. In this example, data from a single machine reporting to the Forwarder will be reported upon.

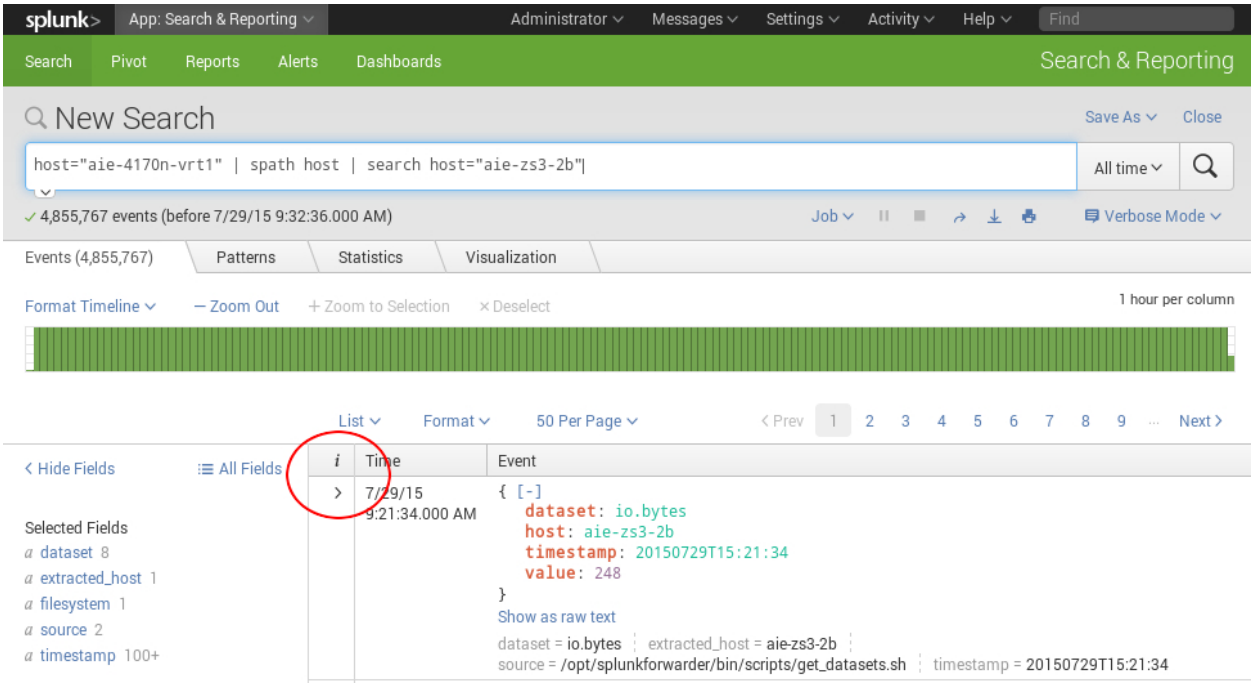

Figure 5. Expanding a record

Note in Figure 5 that under the "***i*** " column there are arrows to expand the record and act upon them. Click on the arrow for the first record.

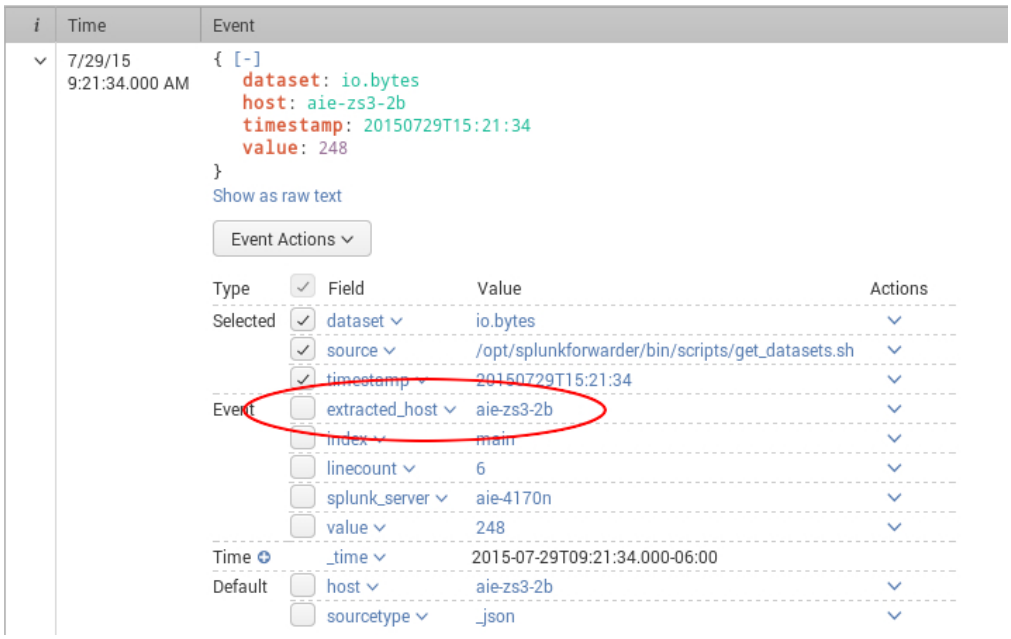

Figure 6. Expanded record

Find the field `extracted_host` in the expanded record and click the box to the left. The field is then moved up to the "Selected" area, making it much easier for the Indexer to sort on a single Oracle ZFS Storage Appliance. This selection is applied to all records pushed from the Forwarder. This action is persistent so it will not have to be repeated for other searches even if they are for different Oracle ZFS Storage Appliances.

Clicking the arrow in the "***i***" column once again will close the expanded window. Note the `extracted_host` field is now displayed as `host` in the log records. By clicking on the value of the extracted host, a list of options is displayed.

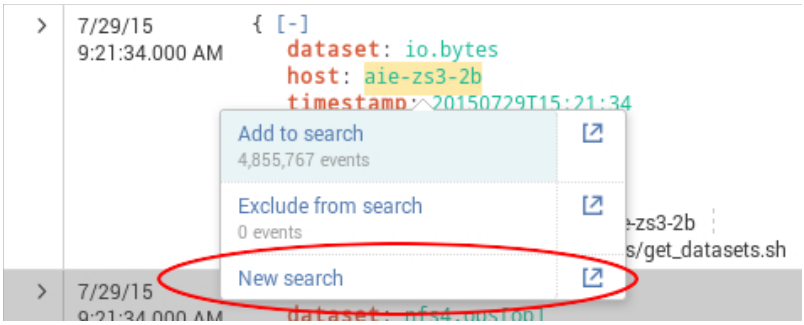

Figure 7. Creating a search

Clicking on "New search" will create a search for only that Oracle ZFS Storage Appliance. Other fields can be added to the search from the same popup menu. By clicking on the dataset value of `nfs4.ops[op]`, the search is restricted to NFSv4 operation events and no other datasets.

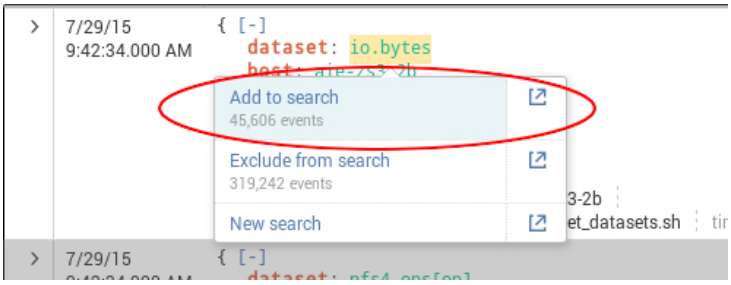

Figure 8. Adding to a search

As the search is created, the command line for it is shown at the top. It may be manually edited if needed.

Example: Charting I/O Operations

As an example of extracting and charting data with drilldown information, a search is created for the desired Oracle ZFS Storage Appliance as shown in the previous figure. You could search through the many events returned for the desired host to find a particular dataset and add it to the search as shown previously, but it is likely faster to simply edit the search manually. Adding an

`spath` command for the dataset field will speed the search up. Then you can pipe the output to a `search` command for the particular dataset of interest – in this case, `io.bytes`:

```
* | spath host | search host="aie-zs3-2b" | spath dataset | search dataset="io.bytes"
```

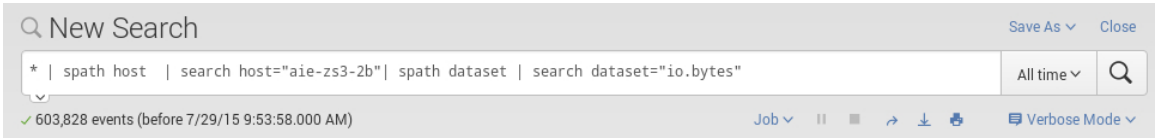

Figure 9. Editing the search

To create a graph from this information, a transforming command[4] can be added to the search.

In the following example, the total I/O in is graphed using the `value` field. Adding `timechart avg(value)` to the `search` command will create a graph with values averaged for each timespan shown, and using '`as "IO Bytes"`' permits renaming the value field for the graph. Use the pull-down menus to change the style and format of the graph. This example shows an area graph for the total I/O in bytes.

```
* | spath host | search host="aie-zs3-2b" | spath dataset | search
dataset="io.bytes" | timechart avg(value) as "IO Bytes"
```

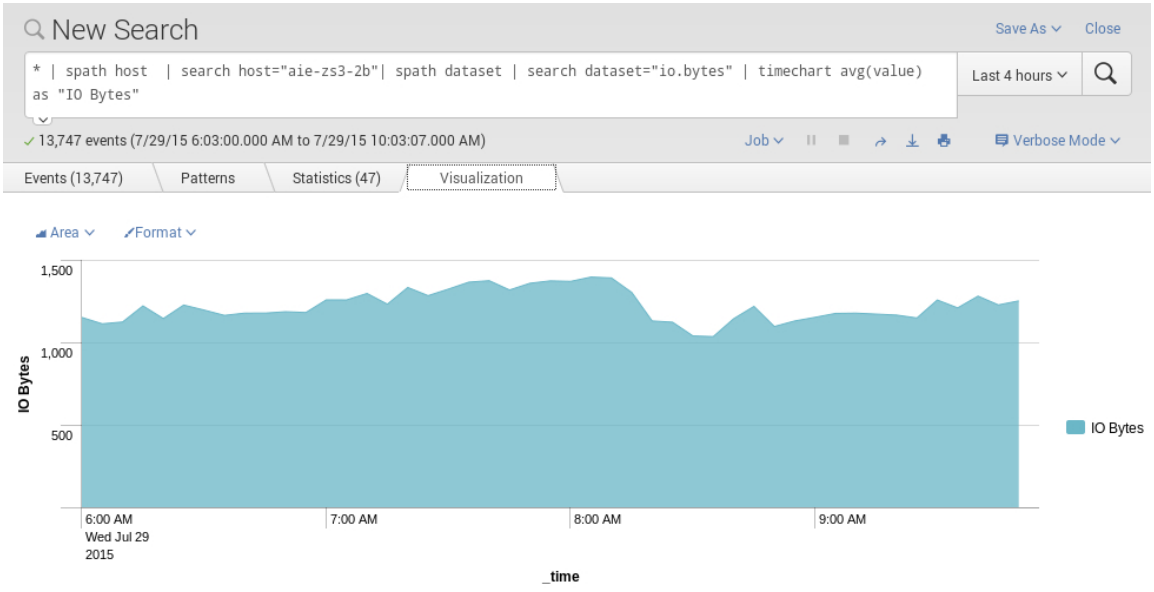

Figure 10. Average total I/O

[4] Transforming commands "transform" the specified cell values for each event into numerical values that Splunk can use for statistical purposes.
Transforming commands include `chart`, `timechart`, `stats`, `top`, `rare`, `contingency`, and `highlight`. Transforming commands are required to transform search result data into the data structures required for visualizations such as column, bar, line, area, and pie charts.

The previous example averages the Oracle ZFS Storage Appliance's I/O every 5 minutes, but this may not represent the data accurately enough, as short spikes of high I/O will be smoothed out in the average. An alternate way to represent the data is to create a table of value over time and use the `chart` command to draw the graph.

```
* | spath host | search host="aie-zs3-2b" | spath dataset | search
dataset="io.bytes" | table value, _time, | chart values(value) as "IO
Bytes/sec" by _time
```

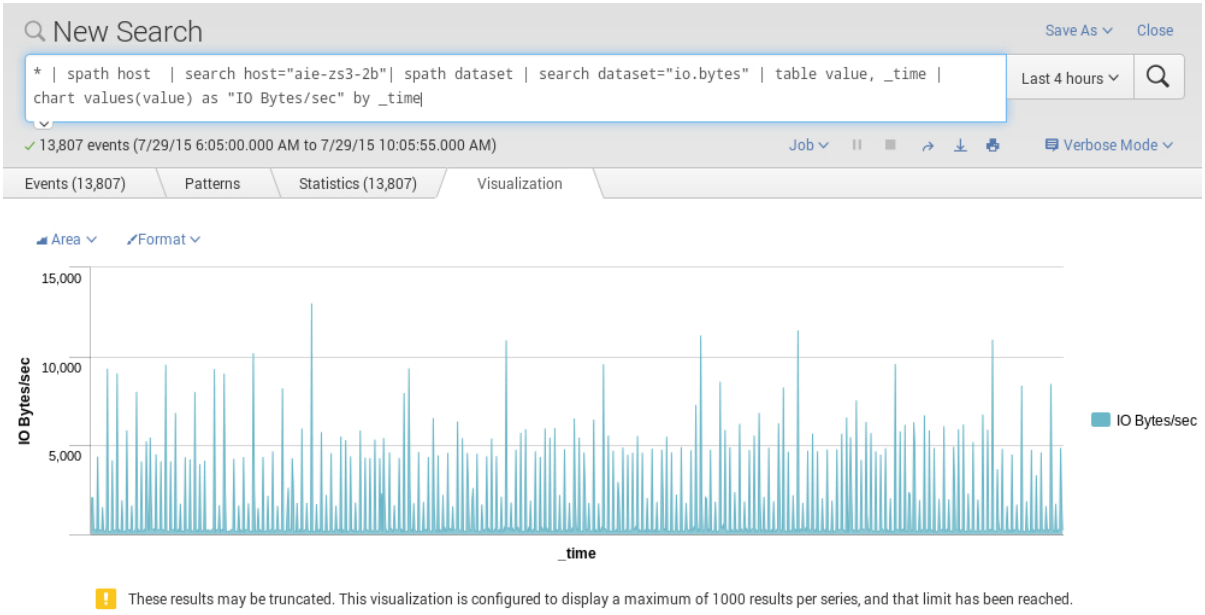

Figure 11. Using the `table` and `graph` commands

The graph in Figure 10 shows clearly that the Oracle ZFS Storage Appliance is averaging about 1,200 B/s in I/O, while Figure 11 shows that there are occasionally spikes of over 10,000 B/s.

Note that there is a warning that there is too much data to graph, and that the results may be truncated. Hovering the mouse over the end of the graph shows that the last time plotted was 1,000 seconds (or 16 minutes and 40 seconds) after the beginning of the graph, with the later data points unplotted. This style of chart is better suited to using a timespan of 15 minutes or less, as shown in Figure 12.

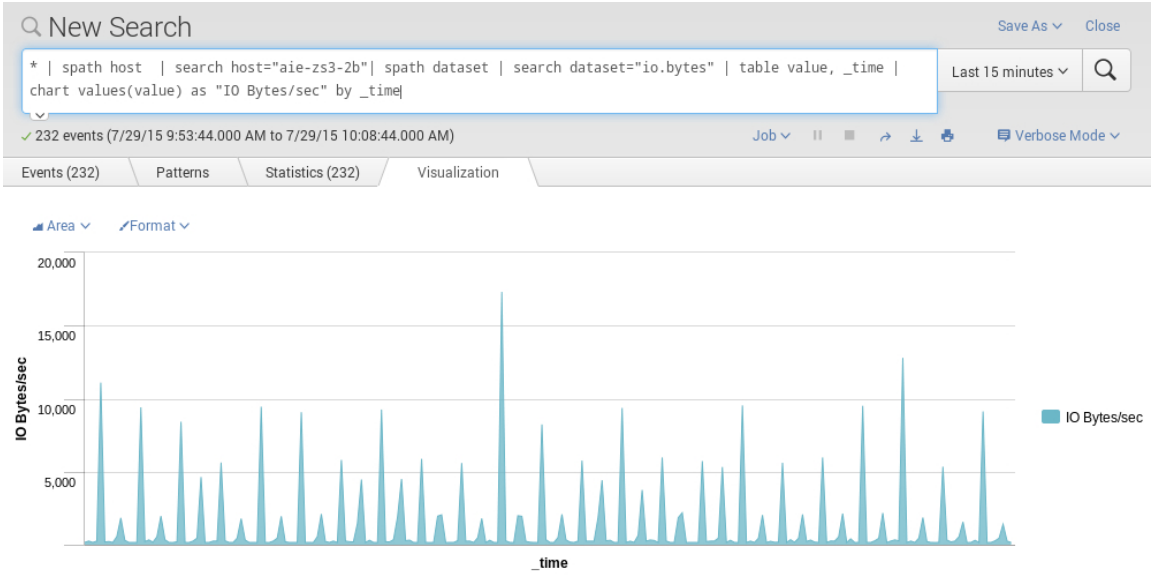

Figure 12. Using a 15-minute timespan

Example: Charting I/O Operations with Drilldown Data

Charting the total I/O of an Oracle ZFS Storage Appliance is useful, but does not tell the whole story. The previous charts using the `io.bytes` dataset provides no clue about whether the load is skewed towards reads, writes, or is a mix of both. To drill down into the total I/O, you can use the dataset `io.bytes[op]` instead. Whenever a dataset's name is followed by a descriptor in square brackets, more information is available. In the case of `io.bytes[op]`, the total I/O shown in `value` is broken down by operation in the fields `read` and `write`, as shown in Figure 13.

Figure 13. `io.bytes[op]` event

By modifying the search used for the chart in Figure 12 to use `io.bytes[op]` as the dataset and modifying the `table` and `chart` commands in the last example to use `read` and `write`, display both pieces of data at the same time, as shown in Figure 14.

```
* | spath host | search host="aie-zs3-2b" | spath dataset | search
dataset="io.bytes[op]" | table read, write, _time, | chart values(read) as
Read, values(write) as Write by _time
```

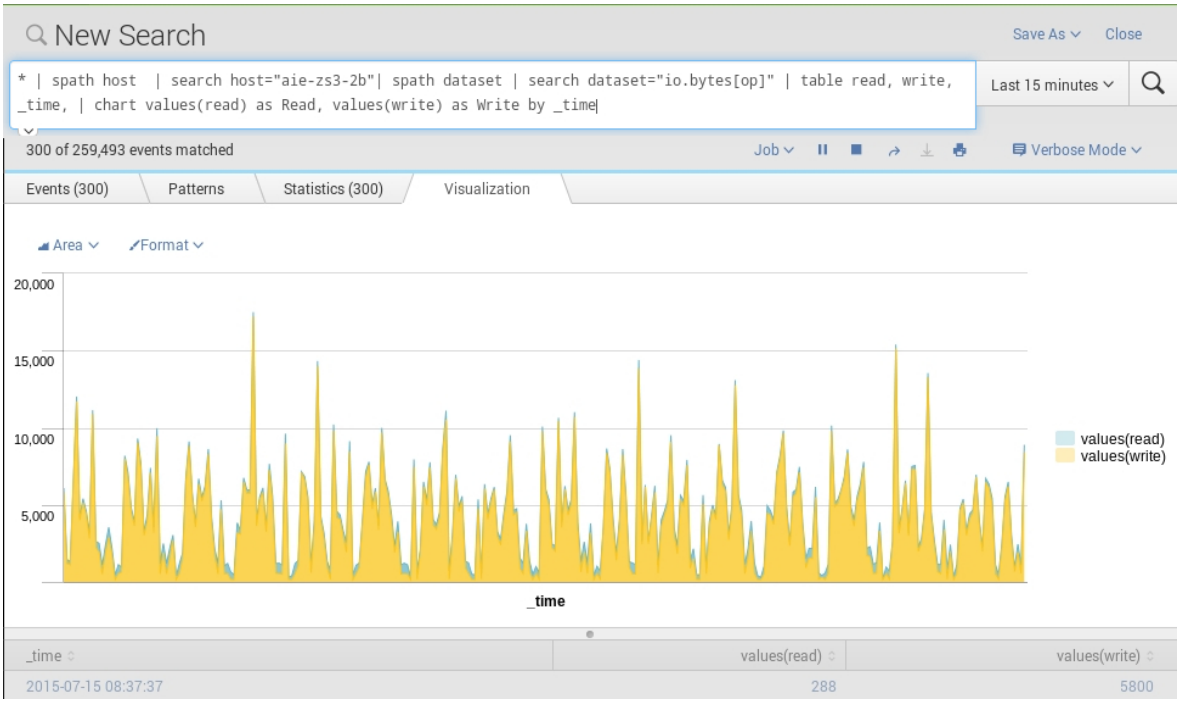

Figure 14. A stacked chart showing read and write I/O

The format of the chart has been set to stack the variables, and now shows the total I/O represented by the total height of the graph, as well as the I/O used by the read and write operations separately. This allows you to easily see that this Oracle ZFS Storage Appliance is handling a much higher write load than a read load.

Collecting Storage Data from the Oracle ZFS Storage Appliance

Storage information may be collected from an Oracle ZFS Storage Appliance in a similar manner to metric information provided by datasets. Once again, you need a script to log on the Forwarder to log in, collect data, process it, and write it to `stdout`. The `get_storage.py` script in Appendix B provides an example.

The `get_storage.py` Script

This script is much simpler than `get_datasets.py`. Its usage is:

```
usage: get_storage.py -k KEYFILE  -u USERNAME
                      zfs_appliances [zfs_appliances ...]
```

All of the security concerns for running `get_datasets.py` remain true for `get_storage.py`, as are the use of the `keyfile` and `username` parameters.

When developing the script to read data from an Oracle ZFS Storage Appliance, it is important to preview the data and ensure that Splunk has the data it needs to properly handle the records. The JSON output for a storage query does not have a host identifier or a timestamp, so it is important that the code add these fields to the JSON object.

The script discards information about the devices that make up the pool and scrubs information for clarity. If this information is valuable to the user, the script should be modified to include it.

Defining the Script

Once again, use a wrapper script:

```
#!/bin/bash
/opt/splunkforwarder/bin/scripts/get_storage.py \
       -k /opt/splunkforwarder/tmp/get_storage.key –u reporter \
       dev-zfs-1 dev-zfs-2
```

Add a script stanza to the file `$SPLUNK_HOME/etc/system/local/inputs.conf` as before:

```
[script:///opt/splunkforwarder/bin/scripts/get_storage.sh]
interval = 60
sourcetype = _json
```

Once the Forwarder is restarted, it will begin executing the script.

Using the Oracle ZFS Storage Appliance Storage Data

The `get_storage.py` script is run once per minute in the example, and returns a JSON object for each pool on the Oracle ZFS Storage Appliance. Here is an example of one such object:

```
{
 "asn": "fd4ccdd9-02e0-c467-90cb-a383680c21c3",
 "available": 958106216960.0,
 "dedupratio": 100,
 "host": "aie-zs3-2b",
 "owner": "aie-zs3-2b",
 "peer": "00000000-0000-0000-0000-000000000000",
 "pool": "pool0",
 "profile": "cache_stripe:mirror:log_mirror",
 "status": "online",
 "timestamp": "20150716T13:57:52",
 "total": 8933531975680.0,
 "used": 7975425758720.0
}
```

A useful visualization from this data is a pie chart of disk space used and disk space available. By creating a search for a desired host followed by a search on the pool desired, you can easily see the ratio of free and used space in the pool.

```
* | spath host | search host=”aie-zs3-2b” | spath pool | search pool=”pool0” | dedup available, used | stats avg(available) as Free, avg(used) as Used | transpose
```

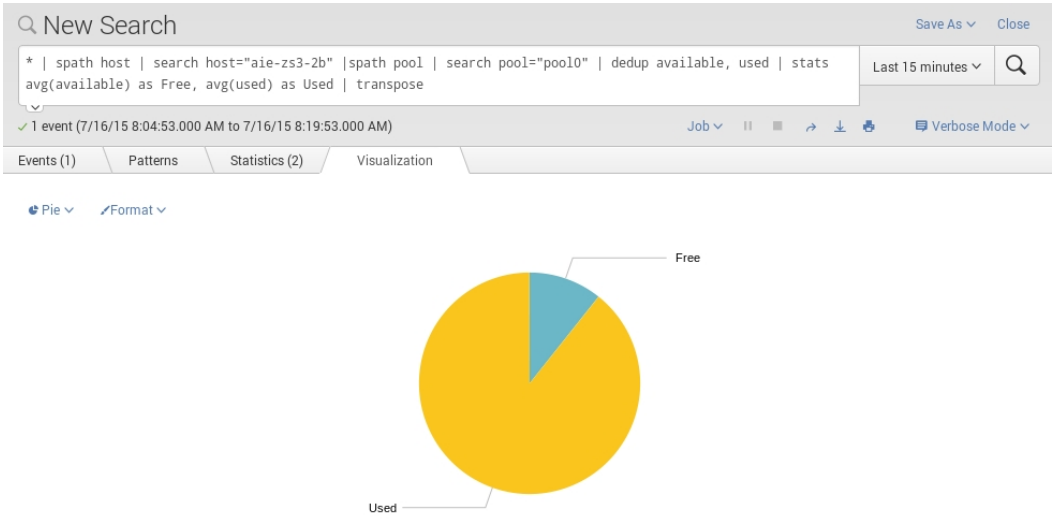

Figure 15. A pie chart showing pool usage

Note the use of the `dedup` command to return only the latest record to be added.

Predictions Using the Oracle ZFS Storage Appliance Storage Data

The `get_storage.py` script in the example can also return information about filesystems. In the code, only filesystems in pool0 have information retrieved.

With the filesystem data being ingested by Splunk, it is a simple process to write a search for the space used in the filesystem.

```
* | spath filesystem | search filesystem="*tinyfs" | timechart
sum(eval(space_total/1048576)) as UsedMB
```

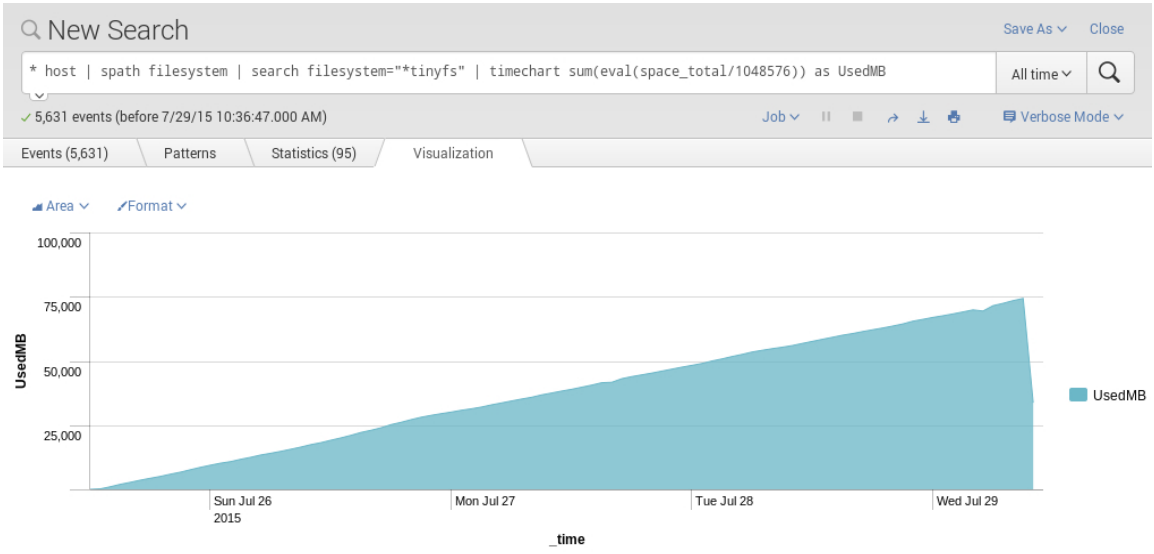

Figure 16. Space used in a filesystem

For this example, there is a script that writes a one megabyte file to a mounted filesystem once every one to three minutes at random, resulting in a steady increase in space used.

To project the space usage into the future, you can use Splunk's `predict` command. This command allows the user to apply one of a number of modeling algorithms to time-based data to predict future trends. Easily invoke the `predict` command by piping the output from the `timechart` command into `predict`.

```
* | spath filesystem | search filesystem="*tinyfs" | timechart
sum(eval(space_total/1048576)) as UsedMB | predict “UsedMB” future_timespan=250
```

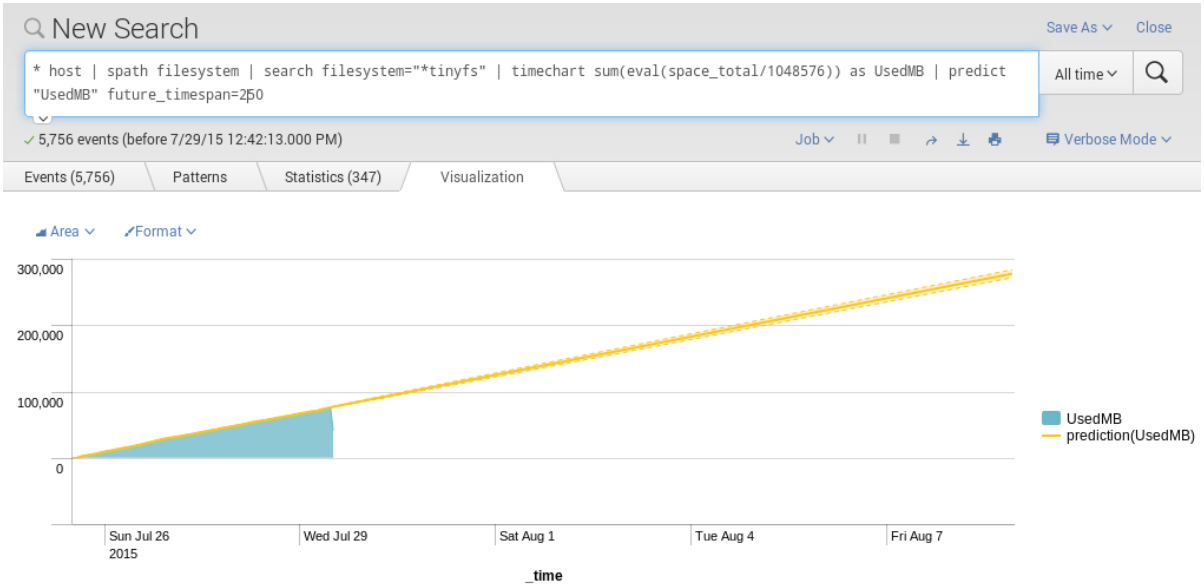

Figure 17. Predicting future space usage in a filesystem

While this example is an oversimplification compared to real-world situations, it clearly shows the power and utility of the `predict` command in Splunk.

Collecting Logs from the Oracle ZFS Storage Appliance

Log files are a rich source of data for Splunk, and the Oracle ZFS Storage Appliance logs a great deal of data. This data is distributed into five different logs: alert, audit, fault, phone-home and system.[5] Only the system log is presented through the standard syslog protocol, which has the ability to forward the system log to another server. By using the REST interface, it is easy to collect all the desired logs.

Three steps are needed to bring the logs into Splunk. The first of these is log collection, which is done with a script.

The `get_logs.py` Script

The `get_logs.py` script that is listed in Appendix C is an example of how the logs may be collected by a script that is designed to be run periodically on a Forwarder node.

The usage of `get_logs.py` as displayed in the script is:

```
usage: get_logs.py –k KEYFILE –u USERNAME
                   -p LOGPATH –t LOGTYPE [LOGTYPE …] [-F]
                   <zfs_appliances>
```

As before, the keyfile should be different for each script. The logpath is the directory where the logs will be written, and one or more logtypes must be specified.

The script, when run, prompts for a password for the Oracle ZFS Storage Appliance username passed on the command line. Again, it is highly recommended to use a non-privileged user to query the REST interface.

The script then authenticates itself to the Oracle ZFS Storage Appliance, gathers the entries for the specified log type that have been generated since the last time the script was run, and writes a file of the form `<zfs_appliance>.<logtype>.log` to a designated path on the Forwarder, rotating out the previous log files in a manner similar to the Linux `logrotate` command.

When the audit log is requested, a special filter can be applied by using the `–F` parameter to prevent writing each session login or logout from the BUI, CLI or REST interfaces. These entries comprise the bulk of the audit file and may not be useful to pass to Splunk for indexing.

One of the issues with collecting the log data through a Splunk Forwarder is that the original source of the data can become conflated with logs from other Oracle ZFS Storage Appliances. In order to be able to search on logs from a single Oracle ZFS Storage Appliance, the script adds a field named “host” to each log entry to correctly identify its origin.

[5] Please refer to the Oracle ZFS Storage Appliance documentation for details on each log type.

The script should be installed in an appropriate location such as `$SPLUNK_HOME/bin/scripts` and be given execute permissions with the command `chmod +x get_logs.py` The designated file path that will be used by the script to write the logs to must exist and be writable by the user running the script.

The script needs to be run on a periodic basis. On Linux and other UNIX-like operating systems, a service called `cron` is available to define schedules to run processes. Adding the following line to the file `/etc/crontab` on the example Oracle Linux system will schedule the log collection process to be run every 15 minutes:

```
0,15,30,45 * * * * /usr/local/bin/get_logs.py –u reporter \
–p /var/log/splunk –k /opt/splunkforwarder/tmp/dev_log.dat \
–t alert audit fault system –F dev-zfs-1 dev-zfs-2
```

With the script in place and set to run on a regular basis, the second step is the configuration of the Forwarder.

The third step that must be in place is to tell the Forwarder to read the logs.

Defining a Monitor

A Splunk Monitor is a process that will watch a specified directory for changes and pass the new data to the Indexer. Because the script in the example writes logs to a file, the Forwarder must be told to monitor that directory for changes to the files located there. The <*filepath*> argument to the `get_logs.py` script becomes the <*source*> for the Monitor.

Any changes to the files in the monitored directory are noted by the Forwarder and immediately acted upon.

To add the Monitor from a file, edit `$SPLUNK_HOME/etc/system/local/inputs.conf` to include a stanza of the format:

```
[monitor://<source>]
sourcetype = _json
```

If the example file path is `/var/log/splunk/`, then the stanza would be:

```
[monitor:///var/log/splunk]
sourcetype = _json
blacklist = \.(gz|bz2|z|zip)$
```

Note that the *sourcetype* must be specified as `_json` or Splunk will improperly parse the logs.

The script does not provide for compressing older log files, because Splunk's default behavior for handling logs is sub-optimal. It understands log rotation to the extent that if a logfile has been renamed, it does not have to be reread, but if the file is renamed and compressed, Splunk will uncompress it and re-parse it, resulting in duplicate events.

If compression is used for the log rotation, adding a blacklist rule as shown in the previous code will prevent Splunk from processing any files ending with the given patterns.

To add the Monitor from the command line, use this:

```
$ splunk add monitor <source> -sourcetype _json
```

Note that the directory name must end with a slash(/) when the command line is used.

To pick up the new configuration, the Forwarder must be restarted with the following command:

```
splunk restart
```

Using the Oracle ZFS Storage Appliance Log Data

As with the dataset and storage data in the previous sections, when the script begins generating data into the monitored directory, the Forwarder will automatically push it to the Indexer.

Example: Charting Logins

You can create a search for login information in a similar manner to the dataset and storage searches. This example uses a wildcard to match all logins to the various interfaces.

```
* | spath host | search host=”aie-7120b” | spath summary | search summary=”User logged in”
```

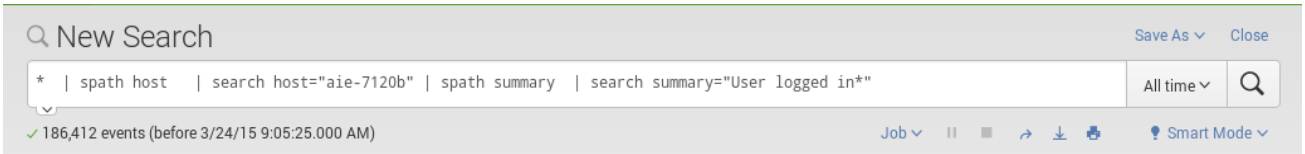

Figure 18. Searching for logins

In the following example, adding “timechart count” to the search command will create a graph of the number of logins over time. Use the pull-down menus to change the style and format of the graph.

```
* | spath host | search host=”aie-7120b” | spath summary | search summary=”User logged in” | timechart count
```

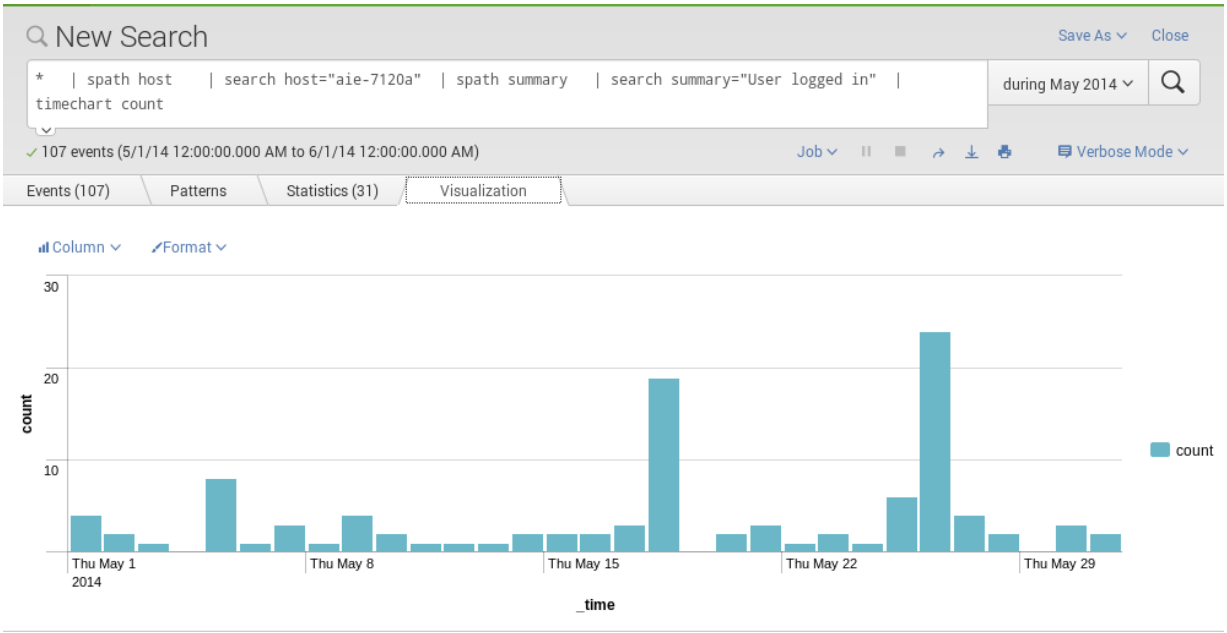

Figure 19. Visualizing the search

Conclusion

By leveraging the RESTful API on the Oracle ZFS Storage Appliance, all logs and datasets can be imported into Splunk, allowing for a comprehensive overview of an organization's infrastructure with the ability to drill down into details as needed.

The provided Python scripts in this paper have been written to illustrate the use of the RESTful API in extracting various types of data from an Oracle ZFS Storage Appliance and may not provide adequate error checking on input parameters or error reporting on failed commands. Please use the examples accordingly.

When creating programs in production environments, pay proper attention to writing code that fully checks user input and provides adequate detail in diagnostic error messages for the user to fully understand the nature of a failure. A more automated method of authenticating the user to the Oracle ZFS Storage Appliance is left as an exercise for the reader.

References

See the following resources for additional information relating to the products covered in this document.

- Oracle RESTful API documentation
  https://docs.oracle.com/cd/E51475_01/html/E52433/index.html
- Oracle ZFS Storage Appliance White Papers and Subject-Specific Resources
  http://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/index.html

  Including: “Working with the RESTful API for the Oracle ZFS Storage Appliance” at
  http://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/restfulapi-zfssa-0914-2284451.pdf
- Oracle ZFS Storage Appliance Analytics Guide
  http://docs.oracle.com/cd/E56021_01/html/E55853/docinfo.html
- Oracle ZFS Storage Appliance Product Information
  https://www.oracle.com/storage/nas/index.html
- Oracle ZFS Storage Appliance Documentation Library, including Installation, Analytics, Customer Service, and Administration guides:
  http://www.oracle.com/technetwork/documentation/oracle-unified-ss-193371.html

  The *Oracle ZFS Storage Appliance Administration Guide* is also available through the Oracle ZFS Storage Appliance help context.
  The Help function in Oracle ZFS Storage Appliance can be accessed through the browser user interface.
- Splunk information and documentation
  http://splunk.com

Appendix A: Python Code for `get_datasets.py`

```
#!/usr/bin/python
# Copyright (c) 2015, Oracle and/or its affiliates. All rights reserved.

""" Collect either a single or a set of data points for each dataset passed via
the command line from each specified ZFS storage appliance and format it in a
way to make Splunk ingestion simple.  If a single data point is specified, it
will be taken for the current second, otherwise the data will be collected
starting from the last time the script was run as stored in the keyfile and be
returned with an object for each one second interval.  If there is no start
time available, collect the last 60 seconds of data.
"""

import json
import sys
import os
import zfs_session


def get_datasets(host, args):
    """ Drilldown dataset data are accumulated over time, and getting a dump
    of the information for a long timespan may take quite a bit of time.
    This function will return the data in one second intervals since the last
    time the script was run against the host (as determined by an entry in the
    key file) up to the present moment. """
    # Get current ZFS appliance's time
    host.get_zfssa_curtime()
    for dataset_name in args.datasets:
        if args.onetime:
            # Override the timespan set in get_zfssa_curtime() to one second
            host.urlspan = "data?start=now&seconds=1"
        host.url = "{0}/api/analytics/v1/datasets/{1}/{2}".\
            format(host.urlbase, dataset_name, host.urlspan)
        dataset_array = host.get_data()['data']
        if args.onetime:
            # A dataset without drilldown data will not be in the list format
            # expected below - recast it into one
            dataset_array = [dataset_array]
        for dataset in dataset_array:
            # Add identifying fields to the JSON object and flatten the
            # object for better ingestion
            dataset['host'] = host.name
            dataset['dataset'] = dataset_name
            dataset = zfs_session.flatten_dataset_json(dataset)
            print json.dumps(dataset, sort_keys=True, indent=2)


def main():
    args = zfs_session.getargs()
    for hostname in args.zfs_appliances:
        # Create the host object
        host = zfs_session.Host(hostname, args)
        get_datasets(host, args)
        host.save_session(args)


if __name__ == '__main__':
    # redefine stdout to be unbuffered
    sys.stdout = os.fdopen(sys.stdout.fileno(), 'w', 0)
    main()
```

Appendix B: Python Code for `get_storage.py`

```
#!/usr/bin/python
# Copyright (c) 2015, Oracle and/or its affiliates. All rights reserved.
""" Collect pool information from a ZFS storage appliance """
import json
import sys
import os
import zfs_session


def get_filesystems(host):
    # Only query for the filesystems in the default project
    host.url = "{0}/projects/default/filesystems".format(host.url)
    filesystems = host.get_data()
    if not filesystems:
        print "No filesystems found"
        sys.exit(1)
    for fs in filesystems['filesystems']:
        fs['filesystem'] = fs.pop('canonical_name')
        # A host and a timestamp are important to Splunk
        fs['host'] = host.name
        fs['timestamp'] = host.timestamp
        # Delete data uninteresting to Splunk
        del fs['source']
        print json.dumps(fs, sort_keys=True, indent=2)


def get_pools(host):
    host.url = "{0}/api/storage/v1/pools".format(host.urlbase)
    pools = host.get_data()
    if not pools:
        print "No pools found"
        sys.exit(1)
    for pool in pools['pools']:
        host.url = "{0}{1}".format(host.urlbase, pool['href'])
        pooldetail = host.get_data()
        pooldetail['pool']['pool'] = pooldetail['pool'].pop('name')
        # A host and a timestamp are important to Splunk
        pooldetail['pool']['host'] = host.name
        pooldetail['pool']['timestamp'] = host.timestamp
        pooldetail['pool'] = zfs_session.flatten_storage_json(
            pooldetail['pool'])
        print json.dumps(pooldetail['pool'], sort_keys=True, indent=2)
        # Only collect filesystem data for pool0
        if pooldetail['pool']['pool'] == 'pool0':
            get_filesystems(host)


def main():
    args = zfs_session.getargs()
    for hostname in args.zfs_appliances:
        # Create the host object
        host = zfs_session.Host(hostname, args)
        host.get_zfssa_curtime()
        get_pools(host)
        host.save_session(args)


if __name__ == '__main__':
```

```
# redefine stdout to be unbuffered
sys.stdout = os.fdopen(sys.stdout.fileno(), 'w', 0)
main()
```

Appendix C: Python Code for `get_logs.py`

```
#!/usr/bin/python
# Copyright (c) 2015, Oracle and/or its affiliates. All rights reserved.

"""Collect logs from an Oracle ZFS Storage Appliance.  The logs are returned in
JSON format and will be rotated as needed."""

import json
import sys
import os
import glob
import zfs_session


def rotate_logs(filename, args):
    """Rotate the log files in a similar manner to the logrotate command"""
    loglist = sorted(glob.glob(filename + '.*'))
    if args.keep_logs == 0 or args.keep_logs >= 1000:
        log_num_len = 3
        args.keep_logs = 1000
    else:
        log_num_len = len(str(args.keep_logs - 1))
    if len(loglist) == args.keep_logs:
        loglist.pop()
    latest = len(loglist)
    for myfile in reversed(loglist):
        logstr = '.' + str(latest).zfill(log_num_len)
        os.rename(myfile, filename + logstr)
        latest -= 1
    os.rename(filename, filename + '.' + str(latest).zfill(log_num_len))
    return


def find_last_logs(filename):
    """Find the last time a log entry was written to the extant logfile and
    return all entries with that timestamp."""
    try:
        with open(filename, 'r') as fp:
            oldlog = json.load(fp)
        fp.close()
        lastlogs = []
        last_time = oldlog[-1]['timestamp']
        for entry in reversed(oldlog):
            if entry['timestamp'] == last_time:
                lastlogs.append(entry)
        return lastlogs
    except IOError:
        return None
    except AttributeError as err:
        print "Error ", err, " opening ", filename
        sys.exit(6)


def get_logs(host, args):
    """ Retrieve the logs from the appliance, parse and filter them as
    required, then write them to the appropriate field. """

    # Get current ZFS appliance's time
    host.get_zfssa_curtime()
```

```
    for logtype in args.logtype:
        filename = '{0}/{1}.{2}.log'.format(args.logpath, host.name, logtype)
        # Find the last log entry written, if any
        if os.path.exists(filename):
            last_logs = find_last_logs(filename)
            if len(last_logs) != 0:
                last_time = last_logs[0]['timestamp']
            else:
                last_time = None
        else:
            last_logs = []
            last_time = None

        if not last_time:
            host.url = "{0}/api/log/v1/logs/{1}".format(host.urlbase, logtype)
        else:
            host.url = "{0}/api/log/v1/logs/{1}?start={2}".format(host.urlbase,
                                                                  logtype,
                                                                  last_time)
        # The appliance returns the logs in a dictionary with a single key:value
        # pair, with the key of 'logs' and a value of a list of all the
        # pertinent entries.  We only need the list.
        log_array = host.get_data()['logs']

        for log_entry in log_array[:]:
            # Using slicing to be able to modify log_array even as we iterate
            # through it.
            log_entry['host'] = host.name
            if args.filter and logtype == 'audit':
                if log_entry['summary'].find("User logged") != -1 or \
                        log_entry['summary'].find("Browser session") != -1:
                    log_array.remove(log_entry)

        # Compare the last log entries written with the new log entries
        # collected and discard any matches in the new log
        for test_entry in last_logs:
            for log_entry in log_array[:]:
                if cmp(test_entry, log_entry) == 0:
                    log_array.remove(log_entry)
                    continue

        if len(log_array) > 0:
            # Only rotate logs and write a new log file if there are new entries
            if os.path.exists(filename):
                rotate_logs(filename, args)
            try:
                with open(filename, 'w') as fp:
                    json.dump(log_array, fp, sort_keys=True, indent=2)
                fp.close()
            except IOError as e:
                print 'I/O error({0}) writing {1}: {2}'.format(e.errno,
                                                               filename,
                                                               e.strerror)


def main():
    args = zfs_session.getargs()
    # Define how many logs are to be kept, up to 1000.  Zero is the same as 1000
    args.keep_logs = 15
    for hostname in args.zfs_appliances:
        # Create the host object
```

```
        host = zfs_session.Host(hostname, args)
        get_logs(host, args)
        host.save_session(args)


if __name__ == "__main__":
    # redefine stdout to be unbuffered
    sys.stdout = os.fdopen(sys.stdout.fileno(), 'w', 0)
    main()
```

Appendix D: Python code for `zfs_session.py`

```
# Copyright (c) 2015, Oracle and/or its affiliates. All rights reserved.
import requests
import json
import sys
import time
import argparse
import getpass
import base64


class Host(object):
    """ Host class for storing ZFS Storage Appliance data and methods for
    authentication and REST data retreival """

    def __init__(self, hostname, args):
        """ Define the host object and create placeholders for variables to
        be set at a later time, then log into the appliance. """
        self.name = hostname
        self.xauth = None
        self.url = None
        self.urlbase = None
        self.urlspan = None
        self.last_access_time = None
        self.client = None
        self.timestamp = None
        self.open_session(args)

    def save_session(self, args):
        """ Copy the configuration for the host to a JSON object and write
        the JSON object to a keyfile.  Note that the keyfile is not intended to
        be shared with other processes.
        """
        try:  # to read the keyfile
            with open(args.keyfile, 'r') as fp:
                tmp_dict = json.load(fp)
            fp.close()
        except IOError:
            tmp_dict = {self.name: {}}
        if self.name not in tmp_dict.keys():
            tmp_dict[self.name] = {}
        # The xauth token should not be stored in clear text.  encode()
        # will use the Vigenere cipher to encrypt it, using the password as
        # a key.
        tmp_dict[self.name]['xauth'] = encode(args.password, self.xauth)
        tmp_dict[self.name]['url'] = self.url
        tmp_dict[self.name]['urlbase'] = self.urlbase
        tmp_dict[self.name]['urlspan'] = self.urlspan
        tmp_dict[self.name]['last_access_time'] = self.last_access_time
        try:
            with open(args.keyfile, 'w') as fp:
                json.dump(tmp_dict, fp, sort_keys=True, indent=2)
            fp.close()
        except IOError as e:
            print 'I/O error({0}) writing {1}: {2}'.format(e.errno,
                                                           args.keyfile,
                                                           e.strerror)

    def open_session(self, args):
        """ Read args.keyfile and if a JSON object exists for the host
        self.name, populate the host object, otherwise define the URL base.
        """
        try:
            with open(args.keyfile, 'r') as fp:
```

```
            tmp_dict = json.load(fp)
        fp.close()
        if self.name in tmp_dict.keys():
            self.xauth = decode(args.password, tmp_dict[self.name]['xauth'])
            self.url = tmp_dict[self.name]['url']
            self.urlbase = tmp_dict[self.name]['urlbase']
            self.last_access_time = tmp_dict[self.name]['last_access_time']
            self.urlspan = tmp_dict[self.name]['urlspan']
        else:
            self.urlbase = "https://{0}:215".format(self.name)
    except IOError:
        # No keyfile at all
        self.urlbase = "https://{0}:215".format(self.name)
    self.define_client(args)

def define_client(self, args):
    """ Define the Sessions object used to authenticate to and interact with
    the ZFS appliance.
    """
    self.client = requests.Session()
    # Do not verify appliance's self-signed certificate
    self.client.verify = False
    self.url = "{0}/api/access/v1".format(self.urlbase)
    # Can we log in with our old key?
    try:
        self.client.headers.update({'X-Auth-Session': self.xauth})
        obj = self.client.get(self.url)
        obj.raise_for_status()
    except requests.exceptions.HTTPError:
        # No self.xauth found, log in with the username and password
        try:
            # Having X-Auth-Session defined in the headers overrides the
            # existence of X-Auth-User and X-Auth-Key
            del self.client.headers['X-Auth-Session']
            self.client.headers.update({'X-Auth-User': args.username})
            self.client.headers.update({'X-Auth-Key': args.password})
            obj = self.client.post(self.url)
            obj.raise_for_status()
            # Remove X-Auth-User and X-Auth-Key and add X-Auth-Session
            del self.client.headers['X-Auth-User']
            del self.client.headers['X-Auth-Key']
            self.xauth = obj.headers['x-auth-session']
            self.client.headers.update(
                {'X-Auth-Session': obj.headers['x-auth-session']})
        except:
            print 'Error with user/pw authenticaton: {0}'.format(
                sys.exc_info()[0])
            sys.exit(2)
    except:
        print 'Error {0}'.format(sys.exc_info()[0])
        sys.exit(3)
    return self

def get_zfssa_curtime(self):
    """ It cannot be assumed that the system running the script will be
    synchronized with the appliances being polled.  Pull one second of arc
    size data from the storage appliance to define our base time.
    """
    self.url = \
        '{0}/api/analytics/v1/datasets/arc.size/data?start=now&seconds=1'.\
        format(self.urlbase)
    nowstr = self.get_data()['data']['startTime']
    nowobj = time.strptime(nowstr, "%Y%m%dT%H:%M:%S")
    nowflt = time.mktime(nowobj)

    # While the timespan is not always needed, the overhead of adding it
    # here is very low and allows the same function to be called for both
```

```
        # single data points and drilldowns.
        if self.last_access_time is not None:
            # Get the data since it was last collected
            lasttime = self.last_access_time
        else:
            # Get last hour's data if being run for the first time
            lasttime = nowflt - 3600
        self.last_access_time = nowflt
        timespan = nowflt - lasttime
        self.urlspan = "data?start={0}&seconds={1}".format(
            time.strftime("%Y%m%dT%H:%M:%S", time.localtime(lasttime)),
            int(timespan))
        self.timestamp = "{0}".format(time.strftime("%Y%m%dT%H:%M:%S",
                                                    time.localtime(nowflt)))

    def get_data(self):
        """ Make the HTTP call using self.url and return a JSON object after
        stripping off the outer ['data'] wrapper """
        # noinspection PyBroadException
        try:
            obj = self.client.get(self.url)
            obj.raise_for_status()
        except:
            print 'Error with REST call: {0}'.format(sys.exc_info()[0])
            print 'URL = {0}'.format(self.url)
            sys.exit(2)
        else:
            return obj.json()


################################################################################
# Below are functions that do not act on the host object.
################################################################################
def flatten_dataset_json(dataset):
    """
    Flatten the JSON object returned by the appliance.  It is assumed that the
    object has already had its outer data wrapper stripped off.
    """
    dataset['timestamp'] = dataset.pop('startTime')
    # Bring the total value to the top level.
    dataset['value'] = dataset['data']['value']
    del dataset['data']['value']
    # Handle breakdown data
    try:
        # The inner dictionaries only exist for datasets for which there
        # are breakdowns available.
        for tmpdict in dataset['data']['data']:
            dataset[tmpdict['key']] = tmpdict['value']
    except KeyError:
        pass
    # Delete unused fields
    del dataset['sample']
    del dataset['samples']
    del dataset['data']
    return dataset


def flatten_storage_json(storage):
    """
    Flatten the storage object returned by the appliance.  It is assumed that
    the object has already had its outer data wrapper stripped off.
    """
    # Try dropping vdev details and other uneeded sub-objects some of which may
    # not always exist in the object
    objlist = 'vdev', 'log', 'spare', 'errors', 'scrub', 'cache'
    for subobj in objlist:
        try:
```

```
            del storage[subobj]
        except KeyError:
            pass
    # Flatten usage data
    try:
        # The inner dictionary 'usage' is brought up a level
        for name, value in storage['usage'].iteritems():
            storage[str(name)] = value
        del storage['usage']
    except KeyError:
        pass
    return storage


def encode(key, clear):
    # Encode a string using the Vigenere cipher
    coded = []
    for idx in range(len(clear)):
        # Rotate through the key string to get the key character
        key_c = key[idx % len(key)]
        # Add the ord() values of the clear and key characters, modulo
        # the size of the character set (256) and append the result to the
        # encoded string
        coded.append(chr(ord(clear[idx]) + ord(key_c) % 256))
    # Ensure the coded string is a pure text string
    return base64.urlsafe_b64encode("".join(coded))


def decode(key, base64str):
    # Decode a string using the Vigenere cipher
    clear = []
    # Ensure that base64str is not a unicode string or decoding from base64
    # will crash
    coded = base64.urlsafe_b64decode(str(base64str))
    for idx in range(len(coded)):
        key_c = key[idx % len(key)]
        clear.append(chr((256 + ord(coded[idx]) - ord(key_c)) % 256))
    return "".join(clear)


def getargs():
    """
    Parse the command line arguments.  Note that this function does not do any
    checking as to whether the arguments are appropriate for the particular
    script being run.
    """
    parser = argparse.ArgumentParser()
    parser.add_argument('-d', '--datasets', nargs='+', type=str)
    parser.add_argument('-k', '--keyfile', type=str)
    parser.add_argument('-u', '--username', type=str)
    parser.add_argument('-p', '--logpath', type=str)
    parser.add_argument('-t', '--logtype', nargs='+', type=str)
    parser.add_argument('-F', '--filter', dest='filter', action='store_true')
    parser.add_argument('-1', '--onetime', dest='onetime', action='store_true')
    parser.add_argument('zfs_appliances', nargs='+', type=str)
    parser.set_defaults(filter=False, onetime=False)
    args = parser.parse_args()
    # The user may wish to automate the password input based on local
    # security concerns; this prompts for a password each time the program
    # is run.
    if not args.password:
        args.password = getpass.getpass(
            "Enter password for user {0}".format(args.username))
    return args
```

Appendix E: Adding a Limited-Privilege User to the Oracle ZFS Storage Appliance

The Oracle ZFS Storage Appliance does not require the user logging into the RESTful API to have any administrative privileges if the user is only reading data, but some areas, such as Analytics, do require special access. A new user with limited privileges should be created to access the RESTful API.

Figure 20. User configuration screen

If Analytics access is desired, begin by creating a new role for the access. Click the plus sign above the list of Roles to get to the Add Role screen as shown in Figure 21.

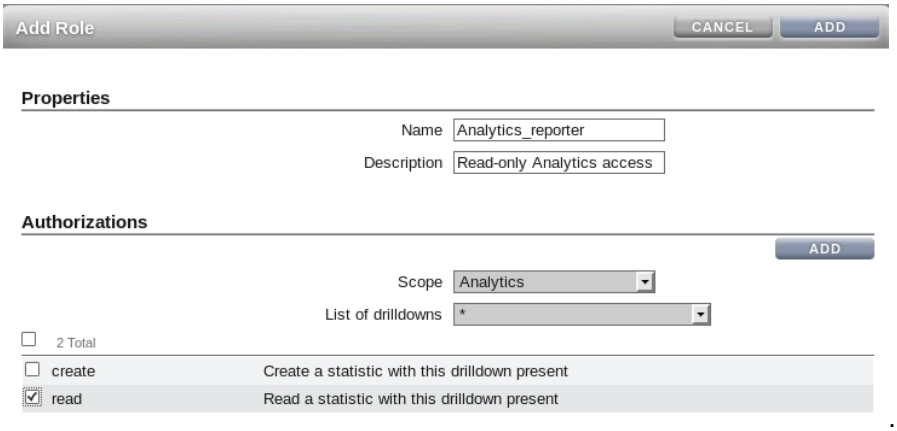

Figure 21. Adding a role

Give the role a name and description, then choose Analytics from the Scope pull-down menu. Check the box for read access and click the Add button in the Authorizations section. The object stat.* with read permission will be listed. You may then click the Add button in the upper right corner to create the role.

To create the user, log in to the Oracle ZFS Storage Appliance's web browser user interface (BUI). Log in as a privileged user such as root. Click on Configuration, then on Users. Click on the plus sign above the list of Users to get to the Add User screen.

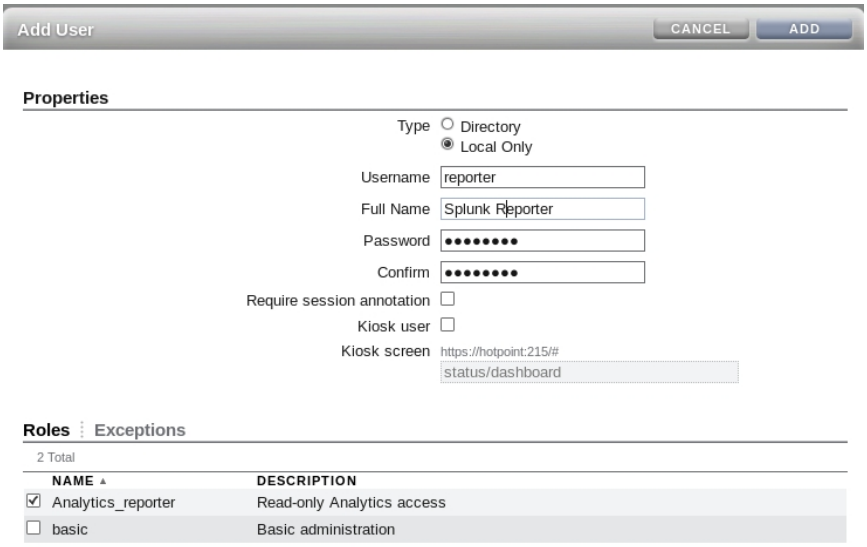

Figure 22. Adding a user

Start by clicking the button to make this user “Local Only”. Enter the name of the new user. This example uses the username “reporter”. Enter a full name for the user if desired, then enter the password twice.

Finally, ensure that the role previously created for access to the Analytics data at the bottom of the window is checked off and that other roles are unchecked. Click the Add button in the upper right corner of the window to add the user to the system.

Appendix F: Enabling the RESTful Interface

In Oracle ZFS Storage Appliance OS8.3 and earlier, the RESTful interface on the Oracle ZFS Storage Appliance must be enabled before it can be accessed. This is done through the browser user interface (BUI). Log in to the BUI and click on the Configuration menu item. The Services page will be displayed. At the bottom of the page is a line in the Remote Access section labeled REST. If the indicator is not green, click on the on/off button as shown in the following figure. This button toggles the REST interface.

Figure 23. Enabling the REST interface

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

ORACLE

CONNECT WITH US

blogs.oracle.com/oracle
facebook.com/oracle
twitter.com/oracle
oracle.com

Integrated Cloud Applications & Platform Services

Splunk and the Oracle ZFS Storage Appliance,
September 2015, version 2.1
Author: Joseph Hartley, Application Integration Engineering