Cray® System Snapshot Analyzer (SSA)

Jeremy Duckworth

Cray Inc.

Table of Contents

Audience

This white paper is intended for information technology professionals. The executive summary is appropriate for information technology managers, while the remainder of the document will best be understood by technical practitioners.

Executive Summary

This white paper provides information about the Cray® System Snapshot Analyzer (SSA). Its purpose is to help Cray customers understand what SSA is, what data it collects, how the data is secured and how the data is used — so customers can make an informed decision on their use of SSA.

What is the Cray® System Snapshot Analyzer (SSA)?

SSA is a Cray customer service application designed to support product issue diagnosis and reduce time to resolution. SSA tracks the configuration of a product over time, identifying and cataloging change. It improves time to resolution by automating:

1) The collection and uploading of product diagnostic information

2) Product health checks

3) The creation of customer support cases

While SSA can run in a local-only mode, automated support features are currently available only when SSA is used with the SSA upload service. The upload service is a standards-based, secure network transfer mechanism responsible for reporting data from a customer system to a Cray datacenter.

Certain use cases of SSA require that it be fully automated. Others require customer intervention, typically in support of a Cray service request (for example, collecting a set of triage information and uploading it to Cray).

Please visit *http://www.cray.com/support/cray-system-snapshot-analyzer* for additional information on the benefits of using SSA.

What data does SSA collect?

SSA collects product-specific configuration, health and diagnostic data (including hardware and software inventories). SSA does not collect information on user files, passwords or application details.

Detailed information on what SSA collects is release specific, and collection documentation is included with every release.

How does Cray use the information collected by SSA?

Generally, Cray uses the data from SSA to deliver proactive product service where possible, and react faster when proactive service is not possible. Cray uses the information to understand product state and configuration at a given point in time, and uses a historical database of information to understand changes in product state and configuration over time.

Please visit *http://www.cray.com/sites/default/files/resources/Cray-SSA-EULA.pdf* for additional information on how Cray uses the data from SSA.

How is data from SSA transferred to Cray?

Data is transferred via the SSA upload service in the form of SSA *snapshots*[1].

The SSA client (the shepherd) establishes a standard HTTPS connection over TCP/IP (TCP port 443) using the Transport Layer Security (TLS) protocol[2]. To establish a TLS connection, the SSA client relies on validation of the upload service at *ssa.cray.com* using a third-party signed X.509 certificate.[3] Once a secure network connection is established, the SSA client uses customer-supplied access credentials (available via CrayPort) to authenticate to the upload service.

The SSA client is connected long enough to transfer any pending snapshots, and then it disconnects. The connection can be initiated only from the SSA client, and no content is downloaded from *ssa.cray.com*.

Introduction

This white paper provides information about the Cray® System Snapshot Analyzer (SSA). Its purpose is to help Cray customers understand what SSA is, what data it collects, how the data is secured and how the data is used — so customers can make an informed decision on their use of SSA. Instructions on installation and configuration of SSA components are not included in this white paper, but a reference is included in the Available Customer Documentation section.

Cray introduced SSA in June 2015 on the Cray® XE6™, XK6™, XK7™, XC30™ and XC40™ supercomputers, and continues to qualify it for use across its product lines[4]. For a year prior to its initial release, SSA was tested in a customer early-access environment. Since the generally available (GA) release for supercomputer products, SSA has processed close to 3,200 unique *snapshots* from Cray customer systems, has detected system issues leading to maintenance actions and has helped accelerate customer case resolution.

The rest of this document provides a list of documentation resources, then covers high-level architecture, followed by a more detailed discussion of the components comprising SSA.

[1] A time-centric collection of information

[2] The SSA client can be configured to use customer network proxy services.

[3] The certificate trust chain is included in the SSA client software, in PEM format.

[4] A release for Cray's Sonexion™ Product is scheduled for Q3 2016

Available Customer Documentation

Please visit *http://www.cray.com/support/cray-system-snapshot-analyzer* for general information on SSA.

Please visit the Cray Customer Documentation Portal at *https://pubs.cray.com* for product-specific SSA installation and configuration guides.

For details on how to activate a SSA customer upload service account and download SSA software for Cray products, please refer to Cray Knowledge Article 4546, *Getting Started with the Cray System Snapshot Analyzer (SSA)*.

If you have questions not addressed in available documentation, please contact a Cray support representative or file a service case against the SSA component[5]. We value your feedback and welcome your questions and suggestions.

Architecture Overview

The SSA architecture can be divided into three areas: 1) the SSA client (the shepherd) and its integration into a Cray product; 2) the SSA client interface to the ssa.cray.com upload service; and 3) private Cray, Inc. "back-end" analytics, user interfaces for Cray employees to access uploaded information and integration into other Cray Customer Service technologies. We provide an overview of the first two areas in this section.

SSA Shepherd Architecture

The SSA client (shepherd) is responsible for performing data collection and information processing tasks. The shepherd is network aware, meaning it is designed to use standard interfaces to collect data across a distributed system, when necessary. The shepherd was designed with heavy consideration toward maximizing operational transparency and minimizing operational impact on customer systems.

The shepherd can be further broken down into two fundamental areas, the core application and shepherd plugins. The core application contains all of the functionality required for configuration, execution and management of the shepherd. Plugins use shepherd core APIs for data collection or information processing tasks – allowing considerable flexibility for operation across sometimes disparate products. They can be grouped into overlapping sets to target very specific data collection and processing tasks – also having programmatic access to data from the current or historical shepherd collection activities.

The shepherd core application is developed using the Python™ Programming Language[6], with one exception. A single executable binary file is included in the shepherd core distribution. This binary is named "isodx," and is utilized only to provide secure network upload functionality. This binary is a Cray-licensed third-party component with corresponding data warehousing facilities in Cray's back-end

[5] Alternatively, you can use the simplified case entry process available via CrayPort.

[6] https://www.python.org/

system. All shepherd plugins are developed using Python, as well. Sometimes helper scripts are developed in an interpreted language (e.g., BASH) and included for plugins to utilize during the collection process.

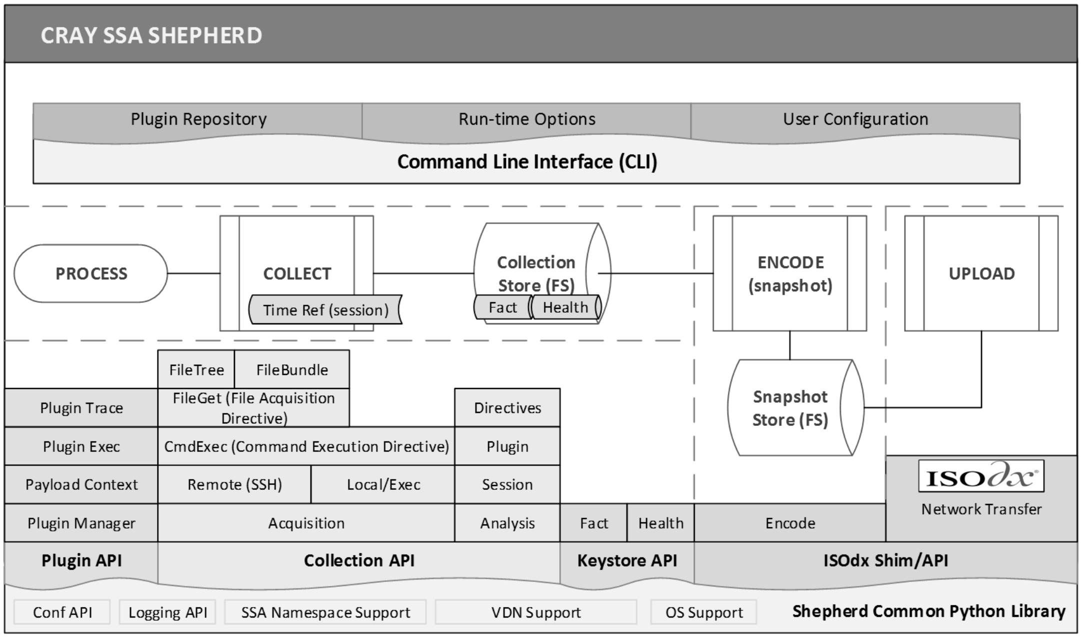


Figure 1. SSA shepherd application architecture

The direct result of executing the shepherd's *collection* stage[7] is a time-centric, file-system output directory tree including a form of the data collected; structured metadata describing what data was collected and details on how data was collected; structured analysis information (health, facts); and one or more human readable collection reports. Self-descriptive, structured data formats are used in the capture of structured information during the process (e.g., JSON[8]) – so all information is wholly contained in the aforementioned time-centric file-system directory. Specifically, the shepherd does not utilize a database service for data storage or retrieval. Indirect results of executing any of the shepherd's processing stages include simple shepherd invocation logs, verbose application-specific logs and auditing and diagnostic information sent to the executing computer's system log facility.

A shepherd plugin *run set* (a group of plugins) forms the input to the *collection* stage. A shepherd *output channel* specifies an output label. With the shepherd, is it also possible to further associate shepherd output with a virtual device name[9]. Together, plugin run sets, output device names and output channels are leveraged for diverse data collection, information processing and reporting activities – all on a single product. For instance, a "health" plugin run set is typically paired with a "health" output channel and run several times an hour to collect, analyze and report system health information. Typical to this use case, there is also a control-focused plugin that only directs the shepherd core to ultimately upload a snapshot if the health of a system has changed.

Time-centric collections from the *collection* stage are encoded into machine-readable, time-centric formats during the *snapshot* stage. All collections for a given output configuration (channel and device) that have not been successfully snapshotted are translated into individual snapshots. The shepherd

[7] The shepherd has three stages of operation: collect, snapshot and upload.
[8] JavaScript Object Notation, http://json.org
[9] Virtual device name (VDN) support was introduced in shepherd core version 1.2.0 to accurately reflect logical identifies within a product

continues to retain the source collection after it has been snapshotted – until a purge policy mechanism removes it.

During the shepherd's *upload* stage, all snapshots for a given output configuration (channel and device) that have not been successfully uploaded, are uploaded. Source snapshots are retained either until they are uploaded or a purge policy mechanism removes them. The upload channel is designed to retry the upload process a number of times in the event network reliability issues are present.

The shepherd core application is installed to one or more central service (e.g., management) nodes within a given environment and performs remote data collection activities against other network nodes, on a platform. The SSA user guide for each Cray platform will detail installation requirements.

SSA Upload Architecture

The SSA client (the shepherd) establishes a standard hypertext transfer protocol secure (HTTPS) connection over TCP/IP, version 4[10], (TCP port 443) using the transport layer security (TLS) protocol [1][2]. To establish a TLS connection, the SSA client relies upon validation of the upload service at *ssa.cray.com* using a third-party signed X.509 certificate.[11] The SSA Upload service does not employ so-called wildcard certificates. The SSA client uses access credentials (available via CrayPort, unique to each customer[12]) once a secure network connection is established to authenticate to the upload service. The credential set the customer uses for upload access control is not the same credential used for data retrieval.

The SSA client is connected long enough to transfer any pending snapshots, and then it disconnects. The connection can only be initiated from the SSA client, and no content is downloaded from *ssa.cray.com*.

The SSA client can also be configured to use a customer-supplied HTTP or SOCKS[13] [3][4] compliant network proxy service[14]. Using an HTTP proxy, the SSA client supports basic [5] and digest [6] authentication[15]. Ports and protocol requirements associated with the direct communication mode are listed in Table 1, with an associated diagram (Figure 2). The same information associated with using a proxyied connection mode are listed in Table 2, with an associated diagram (Figure 3).

[10] IPv6 is not currently supported. Please contact Cray Support if you would like to request support be added.

[11] The certificate trust chain is included in the SSA client software, in PEM format.

[12] Consists of an upload organization and a password. The password is pseudo-randomly generated over a wide character class and is at least 20 characters in length. This password can be reset using CrayPort.

[13] The OpenSSH client and server offer an implementation of the SOCKS protocol.

[14] Properly provisioning and securing network proxy services is not covered in this paper – but there are a number of good, widely available resources for doing so.

[15] Authentication using the SOCKS5 protocol is not currently supported. Please contact Cray Support if you would like to request support be added.

Description	Source Host	Source Port	Destination Host	Destination Port
IPv4 TCP, Port 443, Egress/Outbound Connectivity, HTTPS Application	Customer Cray Host	TCP/1024 or above	ssa.cray.com	TCP/443

Table 1. SSA shepherd communication ports and protocols – no customer proxy

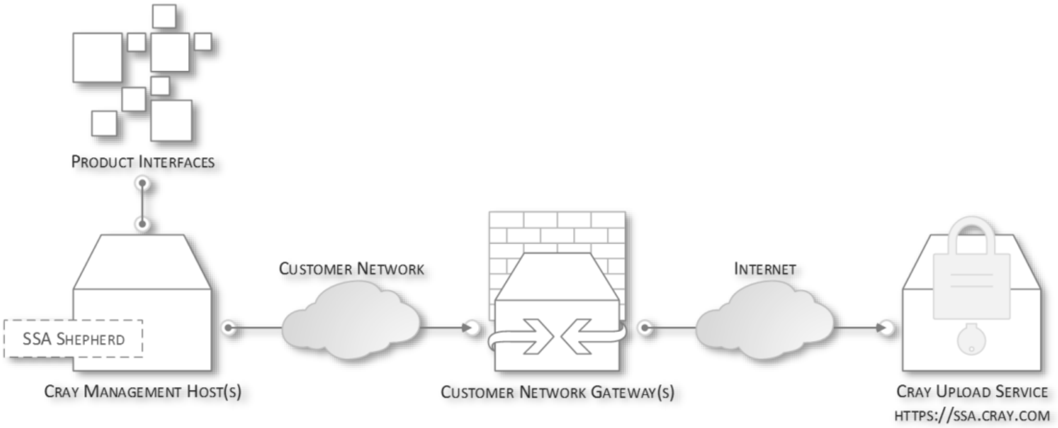


Figure 2. SSA shepherd communication model – no customer proxy

Description	Source Host	Source Port	Destination Host	Destination Port
To Customer Proxy Host				
IPv4 TCP, Egress/Outbound Connectivity to Customer Proxy	Customer Cray Host	TCP/1024 or above	Customer Proxy Host	Customer Defined
To https://ssa.cray.com				
IPv4 TCP, Port 443, Egress/Outbound Connectivity, HTTPS Application	Customer Proxy Host	Customer Defined	ssa.cray.com	TCP/443

Table 2. SSA shepherd communication ports and protocols – with customer proxy

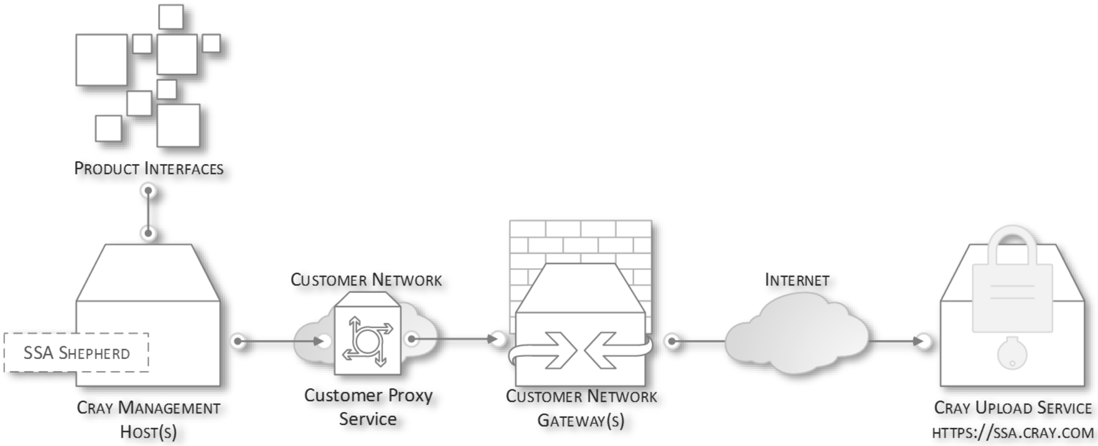


Figure 3. SSA shepherd communication model – with customer proxy

Data Collection Process

In the Architecture Overview section, we introduced how the SSA client (the shepherd) is organized, and introduced the shepherd stages of operation, shepherd plugins, how collection output is organized and the general types of information. In this section, we'll explore the collection process in further detail. Our exploration will cover how the shepherd core discovers, initializes, loads and executes plugins; how a customer can review what each plugin was designed to do; what collection directives are and how they relate to plugins; how to review collection data locally; how to audit shepherd execution; and how the shepherd monitors available system resources before and during execution.

Shepherd Plugin: Interface

Concepts, definitions and relationships involved in the plugin interface are detailed below.

Plugin Namespace

Shepherd plugins are uniquely identified by a hierarchical "namespace." The first label (token) in the namespace represents the highest categorization of plugin topic possible. Each level thereunder represents a further subcategorization (see Figure 4).[16] A plugin's namespace serves many purposes, including:

- The method by which the shepherd loads and validates a plugin
- The method used to designate plugin dependencies within the shepherd
- The runtime methods used that allow a plugin to programmatically "retrieve" data collected from ancestor plugins and previous collection sessions
- Grouping related plugins into common run sets
- Designating an SSA data module (like grouping of information within a collection)
- Reporting runtime information via the CLI, in runtime application logs, etc.

[16] While not covered here, namespaces have syntax grammar rules they must follow

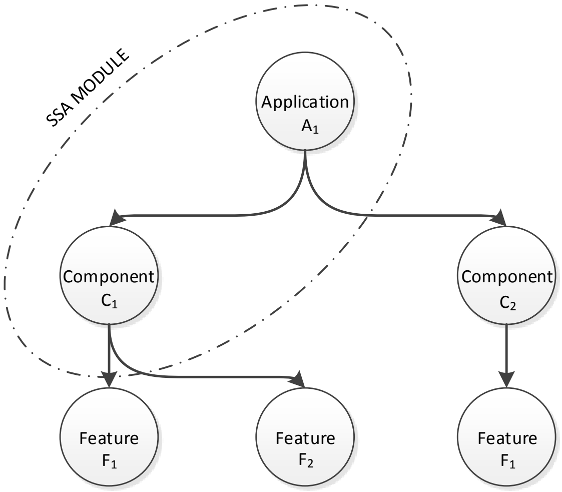


Figure 4. SSA namespace concept

Plugin Configuration Source

Each plugin has its own configuration file, in JSON format. An example configuration file is included in Appendix A.

Plugin Python Source Module

Each plugin has its own Python source module. The module must subclass key Python classes from the shepherd core and contain valid attributes to be executed by the shepherd.

A SHA256 [7] digest, contained in a plugin's configuration file, must match the SHA256 digest at the time a plugin is evaluated for being loaded (not executed) during shepherd runtime. If the SHA256 digest does not match and the shepherd UI functionality to disable SHA256 digest validation is not in place, the shepherd will exit immediately with an appropriate error code and message.[17]

Plugin Compatibility Versioning

Each plugin is required to supply, to the shepherd plugin manager, a plugin_api version string. If the plugin manager does not support a given version, the plugin will not be loaded for execution.

The compatibility information is included both in the plugin's configuration file and its source module – and these two sources must agree for the plugin to be loaded.

Plugin Source Versioning

Each plugin is required to supply a revision string to the shepherd plugin manager.

The plugin revision is included both in the plugin's configuration file, and its source module – and these two sources must agree for the plugin to be loaded.

Plugin Run Levels

Specified in a plugin's configuration file, each plugin must be associated with a run level. The run levels start at 0 and are monotonically[18] increasing.

[17] This is not considered a strong security control since these can be modified locally

[18] e.g., 0, 1, 2, 3, …

Plugin Run Set

Each plugin is associated with zero or more plugin run sets. A run set is an alpha-numeric, textual label that the shepherd uses to select plugins for execution. If the shepherd is not supplied a run set, it looks for plugins tagged with the 'default' run set.

Plugin Dependencies

Each plugin can have zero or more dependencies. Dependencies are specified in terms of plugin namespace and revision. Thus, an exact match must be found (including revision) in the candidate plugin set for the dependency to be satisfied. During execution, if a plugin's dependency exits in an unsuccessful way, the dependent plugin is removed from the execution sequence and marked as having had a dependency fail.

The general process followed by the Shepherd when initializing plugins[19] is:

1. Search for plugin configuration files in the installation defined directory.[20]

2. If the SSA client CLI (ssacli) has options specifying only specific plugins to run, the list of discovered configuration files is pruned to match.

3. The format of and data structure contained within each plugin configuration file is loaded and validated. The name of the configuration and source files, and SSA namespace references within each plugin's configuration file must all agree on a common namespace, or an error is raised and ssacli exits.

4. A plugin run set filter is applied to the set of discovered configuration files. Any plugins not in all of the supplied plugin run sets are pruned. Plugin run sets are configurable via ssacli.

5. Plugin dependencies are validated against the remaining set of discovered plugins. If a dependency is not met, an error is raised and ssacli exits.

6. Plugin source modules are located in the installation defined directory[21], and a SHA256 digest is computed against the source file. If this process fails on any source module, an error is raised and ssacli exits.

After a successful plugin initialization process, the plugins are loaded into a program structure suitable for execution. Each plugin execution stage (initialize, load, run) is distinct.

Shepherd Plugin: Documentation

As covered in the SSA user guide for each Cray platform supported, plugin-level documentation is required by the shepherd plugin API. A README file[22] is included with released software that describes what each plugin in a release was designed to do.

[19] Other shepherd initialization takes place before this process occurs.

[20] Typically `/opt/cray/ssa/default/etc/plugin.d`

[21] Typically `/opt/cray/ssa/default/pkg/libexec/plugins`

[22] Typically located in `/opt/cray/ssa/default/share/doc/README.plugin-doc`

Here is an excerpt from one of the plugin README files:

```
Namespace: cluster.software.rpm
Revision:  1.0
Purpose:   Collect RPM or YUM package information/inventory from all nodes
Node(s):   cluster
Run Set(s): default, triage
```

The actual README will contain a similar stanza for each plugin included in the distribution.

Shepherd Plugin: Collection Directives

Shepherd collection directives are designed to make data collection across a distributed system easier. Collection directives have an object-oriented interface, where a type of directive can be defined, with collection targets "added to" the directive. When all collection targets have been added, a directive is "run" to attempt to perform the desired collection task(s).

Plugin directives can only be utilized within a plugin source module. A directive "context" instance is required to add directives to a plugin, and the context requires a validated plugin configuration reference for successful operation. Thus, the collection directives, plugin manager and plugins co-operate in lockstep.

Collection directives of different types and locuses (see below) can be included in the same plugin.

Plugins do not have to contain directives. Some plugins are not tasked with data collection activities.

There are three primary collection directive types, listed in Table 3.

In addition to the collection directive types, each directive has a locus attribute. The locus attribute is used to determine the underlying execution method. Table 4 introduces the three locus types.

A collection directive of type *CmdExec* can further be classified as a *precondition*. If any collection directives so attributed are included in a plugin[23], they will be executed first and will not store any source data resulting from execution activities (meta-data is still generated from the execution). If a *precondition* fails (does not return configured result), no other directives are run within the specific directive container. *Preconditions* are one method used to guard against the system being in a state or configuration that is not conducive to the collection task.

In all pertinent combinations, there are nine possible ways to configure a collection directive (3 collection types x 3 collection locuses).

Source data collected as a result of collection directives are subject to a series of limits. For *CmdExec* directives, the shepherd will only store up to a configurable STDOUT and STDERR raw size limit (streams are compressed, this occurs while they are being captured). The underlying commands will continue to run, but any output from the streams will not be further recorded (provided additional execution limits are adhered to; see below). For all directives, they are bound by an upper collection limit – typically measured in single-digit GBs or GiBs (most collections in practice are significantly smaller).

Collection directives are also subject to process timeouts. If any plugin directive exceeds a series of progressive timeouts, or the SSA client (ssacli) receives a user-initialed process signal, the directives will attempt to perform an orderly shutdown (using progressively more "powerful" process signals[24]). Timeouts are configured in each plugin's configuration file, and are subject to allowable ranges.

[23] This is slightly abstract, technically multiple CmdExec directives can be instantitated in a single plugin

[24] SIGINT, SIGTERM, SIGKILL and ultimately orphaned as a last resort

Collection Type	Acquires	Generates	Storage Format
CmdExec	STDOUT and STDERR of executed command, attributes of the stored streams	Metadata in JSON format describing the collection activities, in detail Verbose application log data describing collection activities	Indexed and compressed GZIP representations of command STDOUT and STDERR for all CmdExec directives executed within a plugin, by namespace
FileTree	File content and fully qualified path from source host, source metadata describing the file(s) captured	Metadata in JSON format describing the collection activities, in detail Verbose application log data describing collection activities	Exact copies of source files recreated under a collection directory that fully represents the SSA data module, the fully qualified path on the source host and what source host the file(s) were taken from
FileBundle	File content and fully qualified path from source host, source metadata describing the file(s) captured	Metadata in JSON format describing the collection activities, in detail Verbose application log data describing collection activities	A GNU TAR-compatible, GZIP compressed tarball created under the collection directory that fully represents the SSA data module. All structured defined from the FileTree format, above, is contained within the tarball.

Table 3. Shepherd collection directive types

Locus Type	Collection Method
LOCAL	Command(s) are executed or file(s) are acquired locally from the host executing ssacli.
SSH_DIRECT	Command(s) are executed or file(s) are acquired from a remote network host directly accessible via OpenSSH.
SSH_INDIRECT	Command(s) are executed or file(s) are acquired from a remote network host accessible using an intermediate "proxy" host, accessible via OpenSSH.[25]

Table 4. Shepherd collection directive locus types

[25] OpenSSH is used to first login to the proxy node, then the command to be executed is executed on the destination host – again using OpenSSH.

Shepherd Plugin: Collection Introspection

The shepherd can support flexible plugin configurations. These plugin configurations can be designed to build out a collection *workflow* (or *scenario*). Collection directives can be used inside of a plugin for data-collection tasks.

Just as there is a need for data collection in the shepherd, there is also often have a need to inspect the data that has been collected as part of a collection in progress, or a collection session in the past.

Using library APIs in the shepherd, plugins can *open* and programmatically access the data collected as part of any past, successfully executed, plugin's collection directives. Data analyzed in this way can be used to control future plugin execution flow, record information in an available KeyStore (see below) or perform other interesting data translation tasks.

Shepherd Plugin: Collection "KeyStore"

The shepherd includes library APIs that support writing to, and reading from, named keystores. This includes the ability to create a structured health check JSON database and other key-value stores.

The keystore capability is central to the "first-level" analysis capabilities within the shepherd. Namely, we can collect, analyze and then record results using the keystore – where appropriate. The information is then pre-processed for further analysis once the data is uploaded to Cray.

The health database (JSON) is a central interface and data format to enable automation of product health analysis within the SSA ecosystem.

Shepherd: Resource Monitoring and Utilization

As all stages (notably collection) are executed by the shepherd, three key things are monitored: 1) have any signals been received by the parent ssacli process, signaling a progressive shutdown sequence; 2) the instantaneous process load average on the host executing the shepherd; and 3) the collection payload and the actual amount of storage available on the target collection and snapshot file systems. As earlier noted, the shepherd is also responsible for purging off collection and snapshot information based on a set policy. This prevents a long term build-up of SSA output that might otherwise fill up a file system.

The shepherd also drops its parent process *nice* priority to reduce contention with other – more critical – system processes.

Shepherd: Collection Stage Execution

During a collection stage execution, the facilities described above (plugins, directives, introspection, keystore storage) are utilized to collect, analyze and record data about a system. The shepherd supports partial execution of plugins, meaning some plugins can fail while the overall collection survives.

At the conclusion of a successful collection, a time- and device-centric directory is created and populated with the results. A separate installation specific log directory is also updated with simple run logs and verbose shepherd application logs. These logs, along with all of the plugin trace metadata, are also included in uploaded data. This information is very beneficial to assist the SSA team in diagnosing any issues that might arise with SSA operation.

Shepherd: Review Execution Audit Information

For all collection directives, the shepherd utilizes the `sudo` utility.[26] The shepherd utilizes sudo to drop or (dependent upon the shepherd configuration for a given environment) elevate user privileges to a given collection task.

The SSA client CLI (ssacli) and all plugins are executed as a single user, typically root, but defined in an installation-specific shepherd configuration. The directives behave as described above.

Detailed shepherd execution logs, along with simple run logs, are stored in an installation-specific directory.[27]

Since sudo is used for all collection tasks, a detailed audit record is maintained in the log destination for sudo log messages. This source can be used to gain a very granular understanding of data collection activities performed via the shepherd.

Shepherd: Review Collection Data

It is possible for customers to review data locally, to complement existing plugin documentation provided. Appendix B provides one method for local data review.

SSA Client Software Packaging

The SSA client (shepherd) and shepherd plugins are distributed in the binary RPM[28] format. Two RPM packages are typically required, one containing the shepherd core, the other containing plugins and other functionality specific to the supported Cray platform. The SSA user guide for each Cray platform will detail what packages are required.

SSA does not employ any type of mobile code distribution facility via its communication with the server(s) at ssa.cray.com[29].

Securing Information Stored by SSA

Cray has a long history of routinely acquiring and managing the use of product information supplied by customers. We will continue to be diligent and employ reasonable controls in our continued management of product information obtained from our customers.

At a minimum, we will endeavor to adhere to industry best practices for information security in managing the SSA call-home facilities. Our proactive, defensive posture incorporates protection of SSA based on perceived internal and external threats. We will deploy both network- and host-based intrusion detection and prevention system (IDPS) technologies to safeguard SSA call-home facilities. We will design and employ the core principles of safeguarding confidentiality, integrity and availability (CIA) in the continued

[26] https://www.sudo.ws/
[27] See SSA User Guide for specific platform for details on log directory location.
[28] RPM Package Manager (RPM). http://rpm.org/
[29] We do not automatically update the SSA client

operation and maintenance of SSA. We will employ the principles of least privilege and ensure accountability across SSA interfaces.

Conclusion

Thank you for taking time to read the Cray System Snapshot Analyzer (SSA) white paper. The Cray SSA Team hopes that this white paper has better prepared to you make an informed decision on your use of SSA. If questions remain regarding what SSA is, what data it collects, how the data is secured or how the data is used – please contact Cray customer service. We value your feedback and welcome your questions and suggestions.

Cray considers SSA an integral class of technology, enhancing our capability to deliver world-class customer service and support. We invite our customers to participate in the Cray System Snapshot Analyzer program and help guide future development in this area.

References

[1] Rescorla, E., "HTTP Over TLS," RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://www.rfc-editor.org/info/rfc2818>.

[2] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[3] Koblas, D., "SOCKS," Proceedings: 1992 Usenix Security Symposium.

[4] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5," RFC 1928, DOI 10.17487/RFC1928, March 1996, <http://www.rfc-editor.org/info/rfc1928>.

[5] Reschke, J., "The 'Basic' HTTP Authentication Scheme," RFC 7617, DOI 10.17487/RFC7617, September 2015, <http://www.rfc-editor.org/info/rfc7617>.

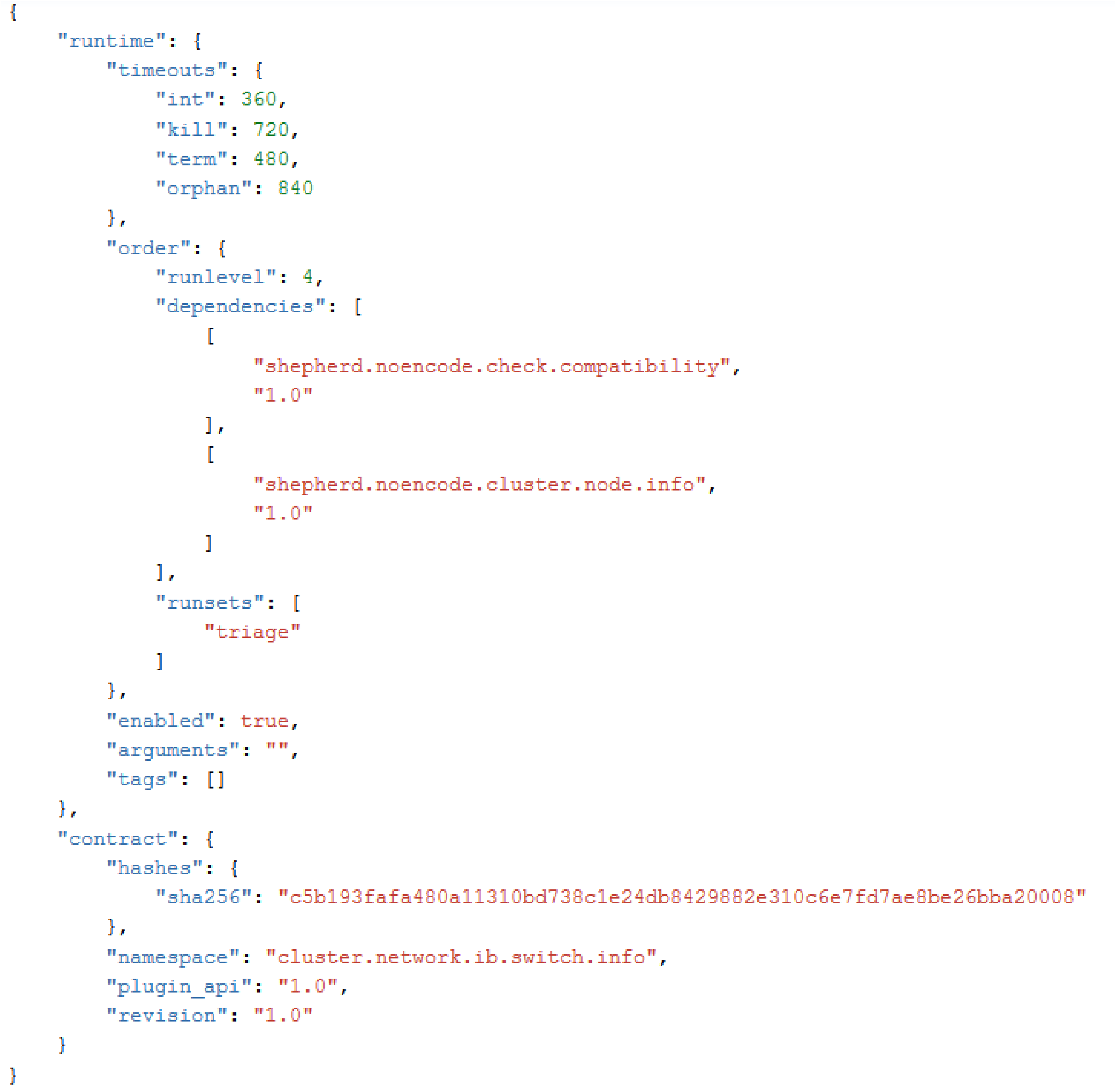
[6] Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication," RFC 7616, DOI 10.17487/RFC7616, September 2015, <http://www.rfc-editor.org/info/rfc7616>.

[7] FIPS, PUB. "180-4." "Secure hash standard (SHS)," March (2012). Last accessed at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf on June 5, 2016.

Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

Appendix A: Example Plugin JSON Configuration File



```
{
    "runtime": {
        "timeouts": {
            "int": 360,
            "kill": 720,
            "term": 480,
            "orphan": 840
        },
        "order": {
            "runlevel": 4,
            "dependencies": [
                [
                    "shepherd.noencode.check.compatibility",
                    "1.0"
                ],
                [
                    "shepherd.noencode.cluster.node.info",
                    "1.0"
                ]
            ],
            "runsets": [
                "triage"
            ]
        },
        "enabled": true,
        "arguments": "",
        "tags": []
    },
    "contract": {
        "hashes": {
            "sha256": "c5b193fafa480a11310bd738c1e24db8429882e310c6e7fd7ae8be26bba20008"
        },
        "namespace": "cluster.network.ib.switch.info",
        "plugin_api": "1.0",
        "revision": "1.0"
    }
}
```

Appendix B: Reviewing Collection Data Locally

Readers can use this appendix as a quick start guide to reviewing information within a collection locally. Information stored in a collection ranges from semi-structured "source" information from the hosting platform, to structured information created by the shepherd, during execution of the collection stage.

As a quick recap for processing stages of the shepherd – there are three: collect, snapshot and upload. The snapshot stage converts raw collections into a machine readable format. Upload securely transfers snapshots back to Cray, Inc.

As prerequisites, you should review and understand the SSA user guide for your Cray platform, the `ssacli (8)` man page and the remainder of this document. This mini-guide is updated as of the Shepherd 1.2.2 release. It is assumed that you have also collected at least one snapshot for the "default" plugin run set and "default" channel.

Please note that some of the output examples have text that line-wraps.

The following topics will be covered:

How to locate the base collection output directory for your installation

How to locate the device and channel specific output directory for your installation

How to locate the collection session directories for your installation

Understanding the files in the top-level collection session output directory

Understanding the sub-directory structure starting with the collection session directory

Locating the metadata folder for an SSA data module

10,000-ft. view of SSA plugin metadata

Locating the data sub-directory for an SSA data module

Examining output of a plugin that included a CmdExec collection directive

Examining output of a plugin that included a FileTree collection directive

Examining output of a plugin that included a FileBundle collection directive

NOTE: Do not alter the contents of any shepherd output. Doing so is not supported and could cause unintended behavior.

How to locate the base collection output directory for your installation

1. Locate your `shepherd.conf` file

2. Document the path assigned to the "collection_dir" configuration option, in the "[collection]" section of shepherd.conf

```
[root@snx11022n000 etc]# pwd
/opt/cray/ssa/default/etc

[root@snx11022n000 etc]# egrep '^\[collection\]|^collection_dir[:=]' shepherd.conf
[collection]
collection_dir: /mnt/mgmt/var/opt/cray/ssa/collection
```

How to locate the device and channel specific output directory for your installation

1. Change to your base output collection directory and list directory contents

```
[root@snx11022n000 collection]# ls -l
total 4
drwxr-x--- 5 root root 4096 Jun 24 12:29 snx11022n000_SNX1600_11022
```

   The single device is `snx11022n000_SNX1600_11022`. This device name is separated, by underscores, into the device name, the product type and the product serial number.

2. Change into your device folder of interest (there can be many, depending on the platform), and list the directory contents

```
[root@snx11022n000 snx11022n000_SNX1600_11022]# ls -l
total 12
drwxr-x--- 9 root root 4096 Jun 25 10:35 default
drwxr-x--- 3 root root 4096 Jun 24 12:29 health
drwxr-x--- 3 root root 4096 Jun 24 12:09 triage
```

   There are three channel directories listed: default, health and triage.

How to locate the collection session directories for your installation

1. Change to the device and channel specific output directory for your installation. In this example, we will use the default channel listed from the last section

```
[root@snx11022n000 snx11022n000_SNX1600_11022]# cd default
[root@snx11022n000 default]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/default
```

2. List the directory contents, reverse-sorting the list by modification time

```
[root@snx11022n000 default]# ls -lrt
total 28
drwxr-x--- 5 root root 4096 Jun 24 12:05 1466787660
drwxr-x--- 2 root root 4096 Jun 24 14:53 1466798003
drwxr-x--- 2 root root 4096 Jun 24 14:55 1466798105
drwxr-x--- 5 root root 4096 Jun 24 15:04 1466798402
drwxr-x--- 5 root root 4096 Jun 24 15:11 1466798831
drwxr-x--- 5 root root 4096 Jun 25 02:34 1466839801
drwxr-x--- 5 root root 4096 Jun 25 10:42 1466868926
```

3. The name of each of the collection session directives is a Unix time EPOCH

```
[root@snx11022n000 default]# date -d@1466868926
Sat Jun 25 10:35:26 CDT 2016

[root@snx11022n000 default]# date -d@1466868926 --utc
Sat Jun 25 15:35:26 UTC 2016
```

Understanding the files in the top-level collection output directory

1. Change into a collection session directory, and list the contents

```
[root@snx11022n000 default]# cd 1466868926

[root@snx11022n000 1466868926]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/default/1466868926

[root@snx11022n000 1466868926]# ls -la
total 204
drwxr-x--- 5 root root   4096 Jun 25 10:42 .
drwxr-x--- 9 root root   4096 Jun 25 10:35 ..
drwxr-x--- 9 root root   4096 Jun 25 10:38 cluster
-rw-r--r-- 1 root root     11 Jun 25 10:39 .collection_complete
-rw-r--r-- 1 root root  14689 Jun 25 10:39 collection_report.txt
-rw-r--r-- 1 root root 151290 Jun 25 10:39 health.json
-rw-r--r-- 1 root root   3111 Jun 25 10:39 instantanswer.json
drwxr-x--- 5 root root   4096 Jun 25 10:39 mgmt
-rw-r--r-- 1 root root   7143 Jun 25 10:39 Pluginresults.json
drwxr-x--- 3 root root   4096 Jun 25 10:39 shepherd
-rw-r--r-- 1 root root     11 Jun 25 10:39 .snapshot_complete
```

2. The `pluginresults.json` file contains a JSON dictionary, with a key entry for each plugin included in the specific plugin run sets

```
[root@snx11022n000 1466868926]# head -11 pluginresults.json
{
    "shepherd.encode.store.health": {
        "deps_failed": false,
        "start_time": 1466869154.7069781,
        "Plugin_api": "1.0",
        "elapsed_time": 0.21535491943359375,
        "stop_time": 1466869154.922333,
        "revision": "1.0",
        "run_level": 8,
        "exitcode": 0
    },
```

3. The `instantanswers.json` file contains a simple JSON-encoded key-value store, if present

```
[root@snx11022n000 1466868926]# head -10 instantanswer.json
[
    {
        "locked": true,
        "values": [
            "{\"1.4.0-28\": [\"snx11022n000\", \"snx11022n001\", \"snx11022n002\", \"snx11022n003\",
\"snx11022n004\", \"snx11022n005\"]}"
        ],
        "description": "JSON-encoded Neo node versions",
        "key": "cssm_node_versions_json",
        "modified": "2016-06-25T15:35:27.268401UTC"
    },
```

Note that there may be additional `*answer.json` files in a collection that serve a similar purpose.

4. The `health.json` file contains a JSON-encoded health database, if present

```
[root@snx11022n000 1466868926]# head -19 health.json
[
    {
        "node": "mgmt",
        "status": "0",
        "healthcheck_version": "1.0.0",
        "description": "Shepherd Neo compatibility check for Shepherd version: 1.2.0\nCompatible Neo
version found, node: snx11022n000, version: 1.4\nCompatible Neo version found, node: snx11022n001,
version: 1.4\nCompatible Neo version found, node: snx11022n002, version: 1.4\nCompatible Neo version
found, node: snx11022n003, version: 1.4\nCompatible Neo version found, node: snx11022n004, version:
1.4\nCompatible Neo version found, node: snx11022n005, version: 1.4\n",
        "title": "Shepherd System Compatibility Check: Neo",
        "severity": "0",
        "creation": {
            "parent_namespace": "shepherd.noencode.check.compatibility",
            "related_namespaces": [
                "mgmt.cscli.cssm.versions"
            ],
            "datetime": "2016-06-25T15:35:27.257802UTC"
        },
        "references": {},
        "id": "health.shepherd.neo.compatibility",
        "uuid": "ca7a3b98-7710-35d4-9004-86e016f852d8"
    },
```

5. While an example is not shown, the `collection_report.txt` file provides a health level view of the shepherd session information, a health summary, an instant answer summary, a high-level plugin trace and configuration and invocation details for the shepherd

6. The `.collection_complete` and `.snapshot_complete` files represent the completion of the collection and snapshot stage, respectfully. These files contain a single line with a Unix EPOCH. This timestamp represents the shepherd session that the stage was performed in. Note that if a collection did not successfully complete, the `.collection_complete` file will not be present – the same follows for the `.snapshot_complete` file. In some channels (notably the “health” channel), you may notice another file named `.collection_bypass`. This file, if set, prevents the snapshot stage from encoding the collection, hence preventing it from being uploaded.

Understanding the sub-directory structure starting with the collection session directory

1. List the directories, two levels deep, in a collection session directory

```
[root@snx11022n000 1466789363]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/health/1466789363

[root@snx11022n000 1466789363]# find .  -maxdepth 2 -type d
.
./mgmt
./mgmt/cscli
./mgmt/cluster
./shepherd
./shepherd/encode
./cluster
./cluster/hardware
```

The names of directories listed is directly related to SSA data modules. The first level of sub-directories represents all unique top-level plugin namespace tokens (e.g., `mgmt`, `shepherd`, `cluster`). The second level thereunder represents all unique second-level plugin namespace tokens (e.g., `cscli`, `cluster`, `encode`, `hardware`). Thus, the `mgmt/cscli` subdirectory structure was created because at least the `mgmt.cscli.fru` or `mgmt.cscli.monitor` plugins contained collection directives in the supplied run sets.

Taken together, the first two tokens of a plugin's namespace form the SSA data module, e.g., `mgmt_cscli` or `mgmt_cluster`.

Locating the metadata folder for an SSA data module

1. Change directory into a fully qualified SSA data module collection directory, and list the contents

```
[root@snx11022n000 1466789363]# cd mgmt/cscli

[root@snx11022n000 cscli]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/health/1466789363/mgmt/cscli

[root@snx11022n000 cscli]# ll
total 8
drwxr-x--- 3 root root 4096 Jun 24 12:29 dstore
drwxr-x--- 2 root root 4096 Jun 24 12:29 mdstore
```

2. Now list the contents of the `mdstore` sub-directory, there will be a single JSON file for each plugin namespace.

```
[root@snx11022n000 cscli]# ls -l mdstore/
total 28
-rw-r--r-- 1 root root 5763 Jun 24 12:29 mgmt-cscli-cssm-versions.json
-rw-r--r-- 1 root root 5841 Jun 24 12:29 mgmt-cscli-fru.json
-rw-r--r-- 1 root root 9889 Jun 24 12:29 mgmt-cscli-monitor.json
```

10,000-ft. View of SSA Plugin Collection Metadata

The JSON-encoded plugin collection metadata stores identified in the last section contain a wealth of information about how a plugin was configured, and how it behaves during execution. This information is considered to be in the "machine readable" category.

It is left as an exercise to the reader to look through one of these files, but you will find the following features:

1. Detail on which collection directive types were present
2. Detail on what the collection directives collected, or attempted to collect
3. Detail on the size of any data captured by the collection directives
4. Paths to the output artifacts of collection directives
5. Execution trace of the execution of each collection directive
6. The full JSON-dictionary of the plugin's configuration file

Locating the data sub-directory for an SSA data module

1. Change directory into a fully qualified SSA data module collection directory, and list the contents of the `dstore` directory (the data store)

```
[root@snx11022n000 1466789363]# cd shepherd/encode/
[root@snx11022n000 encode]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/health/1466789363/shepherd/encode
[root@snx11022n000 encode]# ls -l dstore/
total 8
drwxr-x--- 2 root root 4096 Jun 24 12:29 cmdexec
drwxr-x--- 3 root root 4096 Jun 24 12:29 fileget
```

The sub-directories under the `dstore` directory represent collection directive types. While `cmdexec` is self explanatory, the `fileget` sub-directory covers both the FileTree and FileBundle directive types.

Examining output of a plugin that included a CmdExec collection directive

1. Change directory into a fully qualified SSA data module, data store sub-directory. For this example, we switch over to the `triage` output channel for the example device

```
[root@snx11022n000 ~]# cd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/triage/1466788162

[root@snx11022n000 1466788162]# cd mgmt/cscli/dstore/

[root@snx11022n000 dstore]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/triage/1466788162/mgmt/cscli/dstore
```

2. Change into the `cmdexec` sub-directory, and list the directory contents

```
[root@snx11022n000 dstore]# cd cmdexec/

[root@snx11022n000 cmdexec]# ll
total 28
-rw-r--r-- 1 root root  310 Jun 24 12:09 mgmt-cscli-cssm-versions.err.gz
-rw-r--r-- 1 root root  345 Jun 24 12:09 mgmt-cscli-cssm-versions.out.gz
-rw-r--r-- 1 root root  281 Jun 24 12:15 mgmt-cscli-fru.err.gz
-rw-r--r-- 1 root root 4025 Jun 24 12:15 mgmt-cscli-fru.out.gz
-rw-r--r-- 1 root root  345 Jun 24 12:09 mgmt-cscli-monitor.err.gz
-rw-r--r-- 1 root root 6963 Jun 24 12:09 mgmt-cscli-monitor.out.gz
```

3. In this example, three plugins are represented, `mgmt.cscli.versions`, `mgmt.cscli.fru` and `mgmt.cscli.monitor`. For each plugin, there is a `.out.gz` and `.err.gz` file. These files represent the STDOUT and STDERR collected via collection directives, for all plugins listed. These files are semi-structured, with headers and footers representing specific command outputs.

```
[root@snx11022n000 cmdexec]# zcat mgmt-cscli-fru.out.gz | grep '##@@##!!ENCODEHDR'
##@@##!!ENCODEHDR!!CDR BEGIN: cmdfile /opt/xyratex/bin/cscli fru -n all
##@@##!!ENCODEHDR!!begin 644 /opt/xyratex/bin/cscli fru -n all
##@@##!!ENCODEHDR!!START_RAW::mgmt.cscli.fru::1
##@@##!!ENCODEHDR!!END_RAW::mgmt.cscli.fru::1
##@@##!!ENCODEHDR!!end
##@@##!!ENCODEHDR!!CDR END: cmdfile /opt/xyratex/bin/cscli fru -n all

[root@snx11022n000 cmdexec]# zgrep 'ENCODEHDR.*CDR BEGIN' mgmt-cscli-fru.out.gz
##@@##!!ENCODEHDR!!CDR BEGIN: cmdfile /opt/xyratex/bin/cscli fru -n all
```

As in the example, you can quickly get a sense of what command output was collected by searching for the “CDR BEGIN” lines. Everything between the START_RAW and END_RAW lines represents output from the command. Note that a new line is added to the end of all command outputs to ensure consistent structure.

Examining output of a plugin that included a FileTree collection directive

1. Change into an available `fileget` sub-directory, and list the directory contents, noting that the directory contents represent an SSA plugin namespace.

```
[root@snx11022n000 1466788162]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/triage/1466788162

[root@snx11022n000 1466788162]# cd shepherd/encode/dstore/fileget/

[root@snx11022n000 fileget]# ls -l
total 4
drwxr-xr-x 3 root root 4096 Jun 24 12:16 shepherd-encode-diagnostic
```

2. Change directory into one of the plugin sub-directories, list the directory contents. Note that the sub-directories at this level represent the network host the collection directive was labeled as having collected data from

```
[root@snx11022n000 fileget]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/triage/1466788162/shepherd/encode/ds
tore/fileget

[root@snx11022n000 fileget]# cd shepherd-encode-diagnostic/
[root@snx11022n000 shepherd-encode-diagnostic]# ls -l
total 4
drwxr-xr-x 3 root root 4096 Jun 24 12:16 localhost
```

3. Change directory into one of the host sub-directories, then into the filetree sub-directory, then list the contents. Note that the directory structure until the filetree sub-directory is the same as existing on the capture host. You can access any file, in its native format as captured, in this manner

```
[root@snx11022n000 shepherd-encode-diagnostic]# cd localhost/filetree/

[root@snx11022n000 filetree]# ls -l
total 8
drwxr-xr-x 3 root root 4096 Jun 24 12:16 mnt
drwxr-xr-x 3 root root 4096 Jun 24 12:16 opt

[root@snx11022n000 filetree]# grep -v '^#'
opt/cray/ssa/default/pkg/libexec/plugin/cluster_software_rpm.py | head -10

import json

import lib.Plugin.Plugin as Plugin
import lib.Plugin.directives as directives
import lib.shepherd.keystore as keystore

class PluginDefinition(Plugin.Plugin):

    def __init__(self, **kwargs):
```

Examining output of a plugin that included a FileBundle collection directive

1. Using the same approach as the previous section, locate a filebundle storage sub-directory for plugin (collection directive), and list the directory contents

```
[root@snx11022n000 filebundle]# pwd
/mnt/mgmt/var/opt/cray/ssa/collection/snx11022n000_SNX1600_11022/triage/1466788162/mgmt/logs/dstore/f
ileget/mgmt-logs-collect/mgmt/filebundle

[root@snx11022n000 filebundle]# ll
total 22052
-rw-r--r-- 1 root root 22580868 Jun 24 12:15 filebundle.tgz
```

2. The filebundle.tgz file represents all files captured as part of a FileBundle collection directive for the respective plugin, again containing the full source capture path.

```
[root@snx11022n000 filebundle]# tar tvfz filebundle.tgz
-rw------- root/root        20 2016-06-21 00:03 mnt/mgmt/var/log/nfs-20160621.gz
-rw------- root/root        20 2016-06-20 00:03 mnt/mgmt/var/log/nfs-20160620.gz
-rw------- root/root    207559 2016-06-16 00:03 mnt/mgmt/var/log/ha.log-20160616.gz
-rw------- root/root    208266 2016-06-15 00:03 mnt/mgmt/var/log/ha.log-20160615.gz
-rw------- root/root   8269055 2015-05-31 05:02 mnt/mgmt/var/log/messages-20150531.gz
-rw------- root/root   8291756 2015-05-29 23:02 mnt/mgmt/var/log/messages-20150529.gz
-rw------- root/root 214489026 2015-04-29 18:02 mnt/mgmt/var/log/messages-20150429
```