RSA NETWITNESS® SUITE

Plugins Development Guide

Trademarks

License Agreement

Third-Party Licenses

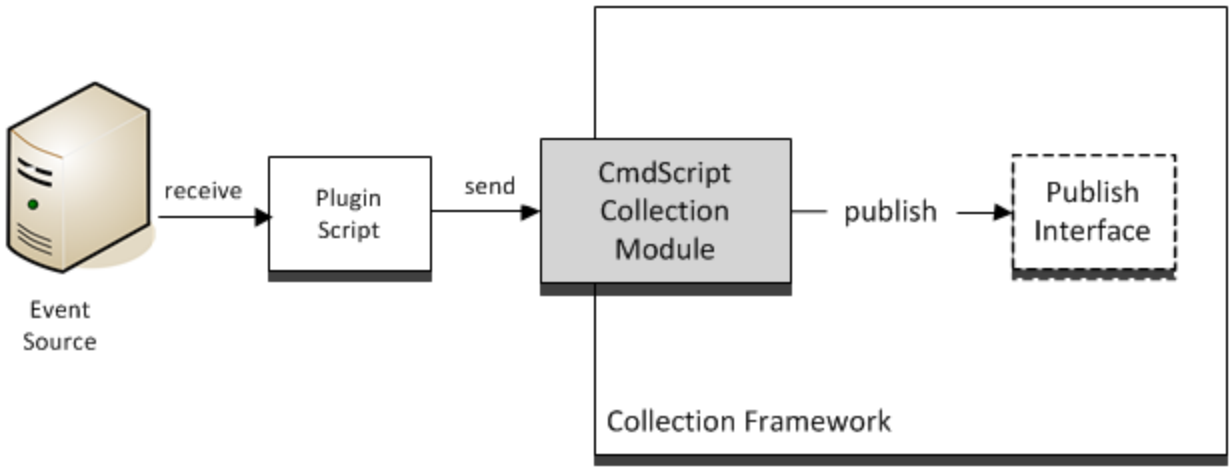
Note on Encryption Technologies

Distribution

Contents

Introduction to the Plugin Collection Developer's Guide

The CmdScript Plugin Collection is a generic collection framework for collecting events using external scripts written in other languages. A set of scripts and associated definition and other required files are packaged as an **Event Source Type**. For example, cloudtrail and azureaudit are both event source types that utilize the CmdScript Plugin collection. The plugin collection's event source types are deployed and managed as logcollection content just like the File, ODBC and other collection event source types. Each plugin collection event source type is referred to as a Plugin.



What is a Plugin?

A Plugin Event Source is a set of definition files and executable files delivered as logcollection content that plugs into the Plugin Collection Module. It is composed of these items:

- Event Source Type Configuration Definition (**typespec.xml**) UI Display Definition (**display_properties.xml**)
- Event Transformation Definition XML File
- Script or Executable + any additional resource files required for collection

Why Use a Plugin?

The major difference between any of the collection types (Checkpoint, File, ODBC, and so forth) is the logic required to retrieve the raw event data. Once the raw event data is retrieved from the endpoint by one of these collection types, the remaining processing that occurs using the Log Collector collection framework is very similar. Often the difficulty in supporting a new Event Source Type is that it does not fit into one of the existing collection types, and requires a different means to retrieve the raw event data. Sometimes this can be solved by pushing the data via syslog or by transferring the data via file collection. When neither of those methods are an option, a Plugin can be used to create a script or program that retrieves raw event data and then pass that event data into the Log Collector via the Plugin Collection. The Plugin Collection handles most of the collection work:

- Manages the configuration of the plugin's parameter data
- Manages the lifetime and execution of the plugin's script processing
- Retains the plugin's persistent bookmark state data
- Manages and retains the plugin's encrypted authorizations and security information

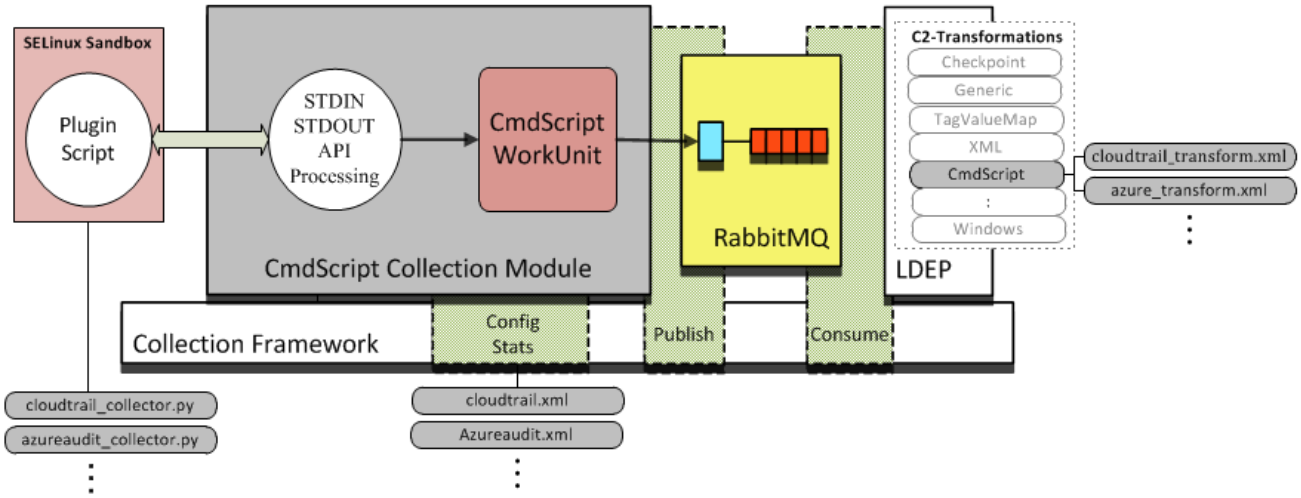
What Are the Benefits of Plugins?

Plugins have the following benefits:

- Delivered completely as Live Content Independent of product release schedules
- Simplifies the creation of new Collection types
- Allows faster time to Market
- Provides support for event sources outside of RSA marketing scope
- Developed by customers, partners, and support engineers and ASOC development
- Can be written in virtually any programing or scripting language
- Integrates seamlessly into:
  - NetWitness User Interface Log Collection Configuration (UI & REST)
  - Log Collection Processing Model
  - Core Logging infrastructure
  - Core Health & Wellness Statistics infrastructure

How Does the Plugin Work?

The following diagram provides an end-to-end view of the Log Collector, including the plugin script.



Note: The Content 2.0 Transform functionality is called asynchronously from the LDEP, and may in the general case be invoked on a separate host.

The basic operation is as follows:

1. The collection module schedules the execution of the work unit on the polling interval.
2. The work unit starts the script in a secure SE Linux sandbox.
3. The work unit passes the configuration for to the script using the **stdin** pipe.
4. The script processes the configuration information.
5. The script connects to the event source and begins retrieving event data.
6. The script responds to the work unit passing events in either tag/value pairs or JSON format using **stdout** pipe.
7. The work unit receives each event through a callback.
8. The work unit processes each event:
   a. Converts event into Name/Value pair format (Event Protobuf)
   b. Adds additional collection meta data
   c. Publishes the event to queue (RabbitMQ)
9. Event processing continues until either:
   - there are no more events to collect, or
   - the workunit requests the script to stop by sending a stop request over **stdin**.
10. The script responds with done signal over **stdout**, and then exits.

Note: Normal execution is between 0-5 minutes.

11. The LogDecoder Event Processor (LDEP) processes each event from the queue (Log Collector only):
    a. The LDEP determines the event source type for the event.
    b. The LDEP loads the corresponding transformation XML for the event.
    c. The LDEP transforms the event into CEF format.
    d. The LDEP sends the CEF event to the Log Decoder.

For more details, see Setup.

How Do I create a Plugin?

These are the basic steps required to write a plugin.

For more details, see Task Checklist and Research.

1. Write the script to perform the collection.
2. Write the script to perform a test connection.
3. Create the XML-based type specification file to define the **Event Source Type**.
4. Define the event source specific configuration parameters.
5. Create the XML-based display properties file used to define the configuration parameters presented in the UI.
6. Create the XML-based transform specification file used to define how the transformation to the CEF format is handled.

Points to keep in mind:

- Development should be done iteratively building upon incremental code improvements.
- Initially, development can be done offline with just the python source. Configuration can be saved in a file and piped into the python.
- Starting initial work on the `on_collect` method in the **<plugin>_collector.py** source file will help flush out the configuration parameters.
    - Start with how to connect to the event source.
    - Move on to how to obtain or get raw event data from the event source.
    - Think about how to translate the raw event information into JSON format if not already in JSON.
    - Think about which fields in the JSON event can be used for the persistence values used to track where you are between process runs. This makes it much easier to control and debug.

  - Use the command line interface for testing, until you are ready to run the plugin through the Log Collector.
- Create the UI definition file **<plugin>_properties_en.xml** and the CEF transformation file **<plugin>_transform.xml** after you have something working.
- Alternatively, adding the UI definition sooner can be a help if you want to test the plugin through the Log Collector.

IMPORTANT: Never develop or test your plugin on a production system.

Task Checklist

This section provides a Task Checklist to guide you through developing your plugin.

Tasks are broken down into the following categories:

- Research
- Setup
- Implementation
  - Configuration
  - Coding Plugin Script
  - Coding Test Connection Script
- Verification

Research Tasks

[ ] Research desired event source

[ ] Determine how to connect to desired event source using python

[ ] Determine how to retrieve event messages from desired event source using python

Setup Tasks

[ ] Prepare a plugin development environment

[ ] Create plugin files using template

[ ] Get Familiar with the plugin source files

[ ] Install additional libraries (optional)

Implementation Tasks

Configuration

[ ] Adding Configuration Parameters

[ ] Exposing Configuration Parameters in User Interface

[ ] Defining CEF Parser Mapping

Coding - Plugin Script

[ ] Adding Connection Logic

[ ] Adding Event Retrieval Logic

[ ] Adding Event Processing Logic

[ ] Adding Bookmarking Data

[ ] Publishing Events to Log Collector

[ ] Adding Logging Output

[ ] Adding Command Line Options

[ ] Adding Configuration Initialization

[ ] Adding Configuration Change Support

Coding - Test Connection Script

[ ] Adding Connection Logic

[ ] Adding Test Retrieval Logic

Verification Tasks

[ ] Verify script exits within 5 seconds when requested to stop by LC

[ ] Verify script correctly handles being terminated by LC

[ ] Verify events are in increasing chronological order

[ ] Verify events parse correctly

[ ] Verify events are bookmarked correctly

Procedures

The procedures in this guide are based on using "Python Script Framework". Using python and this provided python template framework is the recommend method for creating a new plugins. It is also possible and sometime necessary based on the event source's collection API to use other programing languages, but it will require a lot more work, and is not covered in this guide.

See the following sections for details:

Research	Setup	Implementation	Verification and Troubleshooting

Research

Start the process by researching the desired event source.

Research the Desired Event Source

To write a script for collecting from a specific event source, find out as much as possible about the types of devices and resources available, and which events and actions can be collected from them.

In a typical context of cloud infrastructures, devices and resources might be subscriptions, services, virtual machines, users, groups, accounts, and so on. Events and actions might be log-in attempts, or audit-related events (creating, updating, deleting users, permissions, and groups).

- It is very important to research and get an idea of what kind of APIs are available for retrieving different types of events.
- It is very important to outline the requirements to narrow down the plugin to the specific API that you will use.
- Your research should focus on the settings, permissions, and roles that you need to configure on the event source itself.

RSA recommends that you write a proof-of-concept collection script as early in the development process as possible:

- Start with a small client that just verifies how to access a given API.
- Later, retrieve and analyze actual event data.

Connect to a Desired Event Source (Python)

Start by narrowing down the choice of API for the event source to a few candidates (ideally only one). Then, develop a proof-of-concept (POC) client to connect to the API. Typically, a cloud-based event source requires obtaining authorization first (often using a dedicated authorization API), before later requests can be made to the actual API (which is often a different API).

If necessary, a REST client could be used very early in the development cycle to make POST/GET requests to the given API.

However, we recommend to switch to a POC collection script (preferably in Python) as early as possible. Especially, since some event sources provide specialized libraries for Python (e.g. the **adal** library for Microsoft Azure, the **boto** library for AWS) to authorize and access corresponding APIs. If no dedicated library or SDK is available for the event source, use REST libraries such as **requests** in Python.

Retrieve Events From Desired Event Source (python)

After you have developed a Proof-of-Concept (POC) script, and the authorization and connection to the event source is in place, an important part of development is to actually retrieve events from the event source.

In typical cloud-based event sources, this is usually done via REST calls (sometimes implemented in a dedicated library or SDK). Events are then accessible in a number of different ways, including:

- File-based storage accounts (files in formats like XML, JSON, and so on)
- REST calls (GET requests) that return events directly (often in JSON format)
- REST calls that return URLs where the actual events can be accessed indirectly

Some cloud-based event sources may provide "subscription" or "notification" services that notify when events are ready. However, in our context, we recommend to let the script "poll" the event source for events that are available within the given date and time range. This works better within our framework of calling the collection script repeatedly in given **polling intervals**.

It is *very* important to make sure that events are collected in chronological order and without delivering duplicates! This may mean, in some cases, that events obtained for given time "windows" might have to be first stored and then later to be reversed (if obtained in reverse chronological order). If events are not collected in any particular order at all, sorting the events by the corresponding timestamp might be necessary. If this the case, you must limit the number of events that are sorted at one time, in order to keep processing time low.

In some cases, the API may allow to sort by a time stamp via a query parameter in the REST call. Make sure that the time stamp that is used for filtering and sorting is directly related (preferably the same as) the time stamps that events have in the response.

In some cases, we have found that an API uses time stamps for filtering, but returns events that use different time stamps. This can result in some events that are out of order between separate requests. If there is no other way of obtaining events, then a small percentage of events that are out of order may be tolerable. However, the overall order of events should still be in chronological order.

Setup

Prepare a Plugin Development Environment

Preparing a proper development environment will make developing plugins much easier and faster. RSA recommends that you use a non-production NetWitness test system composed of at least a NetWitness UI system node and a NetWitness LD-LC system node to create and test your plugin.

Although it is possible to edit and debug the python logic on the LD-LC system node, you should instead use an offline development environment with an Python IDE for the initial development and testing, along with testing the plugin on LD-LC system. Items such as configuration will be easier on the NetWitness test system, while python coding and debugging will be easier offline.

Remember that if you code and debug offline, you will be in a less strict environment compared to SE Linux. Thus, we recommend you code and debug both offline and on your NetWitness test system.

Recommended steps:

- Setup or use an existing non-production NetWitness test system minimally consisting of the following:
  - NetWitness UI system node
  - NetWitness LD/LC system node
- Create a plugin from a template on a non-production NetWitness test system. For details, see Create Plugin Files using a Template (Python Script Framework).
- Copy the newly-created plugin directory to an offline IDE system. Using a program like scp, pscp, or WinSCP will make this task easier and will also make it easier to sync the two locations.
- Set up the plugin project on the offline IDE system. For an example, see the PyCharm Community Edition development tool section.

List of Template Files

The `/etc/netwitness/ng/logcollection/content/collection/cmdscript/plugin_template` directory on the LogCollector node contains the Python template files. Use the template file to extend the LogCollector with an additional collection protocol method utilizing python 2.7.5 or python 3.3.2. The next sections describe the files associated with the plugin template.

base_plugin.py

This is the base python plugin framework logic file: do not modify this file.

plugin_template_collector.py

Contains plugin specific python logic for the event source. This is the entry point for the plugin, and is called from LogCollector's CmdScript **Plugin** Collection protocol module. This file contains the callback entry points for doing the collection. You will be editing this file, and may also be creating additional source files for your plugin, depending on the plugin requirements.

plugin_template.xml

Typespec file that defines configuration parameters that would be used by LogCollector for new plugin collection. You will be editing this file.

plugin_template_properties_en.xml

The Display Properties definition file. This file defines the how the configuration parameters you have defined in Typespec File are rendered in the UI event source configuration page. You will be editing this file.

plugin_template_transform.xml

The CEF Log Format Transformation definition file. This file defines how the collected event data is transformed into the CEF format log. You have the option of both cherry picking fields, as well as transforming name and value fields. You will be editing this file.

start.sh

The start-up file used to invoke the plugin from Log Collector. This template supports using either python 2.7.5 or python 3.3.2.

template_input

This is a sample input file. Use it to test the plugin outside of the LogCollector Service. It contains configuration input that would normally be passed to the plugin script by the LogCollector's CmdScript **Plugin** Collection protocol module.

test_run_plugin.sh

Script used to test the plugin from the command line in conjunction with the **template_input** file.

create_new_plugin.sh

Script used to automate the creation of a new plugin.

remove_plugin.sh

Script used to automate the removal of a plugin's associated files.

Create Plugin Files using a Template (Python Script Framework)

These steps outline the procedure for creating a plugin from the template files.

Step #1 Create new plugin from template

IMPORTANT: The files in the **plugin_template** directory should never be modified: instead, copy the directory to a new plugin directory using the **create_new_plugin.sh** script.

Usage: `sh ./create_new_plugin.sh plugin_name "Pretty Name"`

Where:

- ***plugin_name*** is the plugin typespec name. This name should be 4-28 lowercase characters, using underscores as necessary to separate words. This is the "typespec" name value similar to other collection protocols (e.g. File or ODBC). It must be a unique name in the plugin typespec namespace.

  Examples: cloudtrail, azureaudit, office365audit, googlecloudaudit

- ***Pretty Name*** is the descriptive name for the plugin, and can include uppercase and spaces. This value must be quoted.

  Examples: "CloudTrail Audit", "Azure Audit", "Office 365 Audit", "Google Cloud Audit"

This script creates the following items:

- a new directory, `/etc/netwitness/ng/logcollection/content/collection/cmdscript/plugin_name` that contains the base starting point for a new plugin
- a new typespec XML file, `/etc/netwitness/ng/logcollection/content/collection/cmdscript/plugin_name.xml`
- a new event transform XML file, `/etc/netwitness/ng/logcollection/content/transform/cmdscript/plugin_name.xml`

Step #2 Test the New Plugin

Test the plugin using the **test_run_plugin.sh** script.

At this point, you should be able to run the python executable and verify if the changes you made in step #1 are complete.

Usage: `sh ./test_run_plugin.sh`

You output should look similar to the following:

```
<LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector starting PluginTemplateCollector
version 1.0.. .<<LOG_INFO> <LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector waiting for
config on STDIN...<<LOG_INFO> <LOG_DEBUG>>2016-07-28T21:12:02Z PluginTemplateCollector waiting
for config...<<LOG_DEBUG> <LOG_DEBUG>>2016-07-28T21:12:02Z PluginTemplateCollector received
config!<<LOG_DEBUG> <LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector
PluginTemplateCollector COLLECTOR-PLUGIN-CALLB ACK 'on_config_init' called<<LOG_INFO> <LOG_
DEBUG>>2016-07-28T21:12:02Z PluginTemplateCollector The phones are working again!<<LOG_DEBUG>
<LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector received config!<<LOG_INFO> <LOG_
INFO>>2016-07-28T21:12:02Z PluginTemplateCollector received bookmark: {u'some_parameter':
5}<<LOG _INFO> <HEARTBEAT>> <LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector starting
run...<<LOG_INFO> <LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector
PluginTemplateCollector COLLECTOR-PLUGIN-CALLB ACK 'on_collect' called<<LOG_INFO> <LOG_
INFO>>2016-07-28T21:12:02Z PluginTemplateCollector starting event processing loop, #events
expect ed: 50<<LOG_INFO>
<LOG_INFO>>2016-07-28T21:12:02Z PluginTemplateCollector setting up connection here...<<LOG_
INFO>
<EVENT format=json version=1 size=220>>{"eventRaw": "{\"starDate\": 73617.3, \"type\":
\"personal log\ ", \"severity\": 3, \"number\": 0}", "bookmarkMeta": {"recordNum": 0,
"lastModified": "2016-07-28T21:1 2:02.390000Z"}, "collectionMeta": {"logType":
"event"}}<<EVENT>
<EVENT format=json version=1 size=221>>{"eventRaw": "{\"starDate\": 73617.3, \"type\":
\"captain's log
\", \"severity\": 3, \"number\": 1}", "bookmarkMeta": {"recordNum": 1, "lastModified": "2016-
07-28T21: 12:02.452000Z"}, "collectionMeta": {"logType": "event"}}<<EVENT>
<EVENT format=json version=1 size=221>>{"eventRaw": "{\"starDate\": 73617.3, \"type\":
\"captain's log
\", \"severity\": 3, \"number\": 2}", "bookmarkMeta": {"recordNum": 2, "lastModified": "2016-
07-28T21: 12:02.514000Z"}, "collectionMeta": {"logType": "event"}}<<EVENT>
<EVENT format=json version=1 size=221>>{"eventRaw": "{\"starDate\": 73617.3, \"type\":
\"captain's log
\", \"severity\": 3, \"number\": 3}", "bookmarkMeta": {"recordNum": 3, "lastModified": "2016-
07-28T21: 12:02.577000Z"}, "collectionMeta": {"logType": "event"}}<<EVENT>
```

Note: If this step doesn't work, you may have an older 10.6.x version of the **plugin_template** directory, which doesn't enable the SE Linux environment correctly. In this case, you can go to step #3. However, RSA recommends that you get the latest version of the plugin_template directory.

Step #3 Verify the New Plugin

Use the NetWitness UI and Log Collector to verify the plugin.

At this point, you should be able to also create the new plugin type from within the Log Collector's event source configuration.

- If the new plugin type is not visible in the Log Collector's event source configuration screen, restart the logcollector service to load the new plugin typespec. This can be done by using the following shell command:

  ```
  systemctl restart nwlogcollector
  ```

- Create an event source instance of the new plugin type by running the **Test Connection**.

- Set Debug to a value of 1 with the new event source instance enabled and CmdScript collection type started.
- You now should be able to view the status of the new event source instance in the Log Collector's log data under the NetWitness **UI Admin > Services > Log Collector Service instance > Logs**. You can also view the event source's statistics using the Explorer view.

Get Familiar with the Plugin Source Files

Look through the following source files, making sure to read the comments, to get an understanding of the logic and requirements.

- <plugin>.xml
- <plugin>_collector.py
- <plugin>_properties_en.xml
- <plugin>_transform.xml

Install Additional Libraries

Add additional libraries and files that are required by your plugin to collect information from the event source into your <plugin> directory.

- All additional files, directories, libraries, etc. need to be installed within your <plugin> directory to be accessible from your plugin executable. Library files should be installed in a **lib** sub-directory of your <plugin> directory.
- Libraries can either be added directly or via **pip** and **virtualenv** (optional).
- If you use **pip**:
  - Create a **requirements.txt** file that contains a list of libraries in your <plugin> directory.
  - Run the following command:

    ```
    bash pip install -r requirements.txt --target ./lib
    ```

    Refer to https://pip.pypa.io/en/stable/reference/pip_install/#options for more information on using **pip**.

Implementation

For configuration, see the following sections:

- Adding Configuration Parameters
- Exposing Configuration Parameters in User Interface
- Defining CEF Parser Mapping

Adding Configuration Parameters

Based on the research you have done so far, you may have list of required and optional parameters you want to configure to connect and pull logs from your event source. All parameters you want users to configure need be added to the <plugin>.xml file, inside the <parameters> block. Here is a sample <parameter> entry in the typespec file:

```
<parameters>
<!-- Example -->
<!-- <parameter> -->
<!-- <nodename>username</nodename> : parameter name, nodename can contain alphanumeric characters and '-' or '_'. -->
<!-- <nodetype>standard</nodetype> : There are 2 types of node. standard(input is visible) and encrypted(input characters will show up as **** instead of visible characters, encrypted type is mostly used for password field). -->
<!-- <prettyname>User Name</prettyname> : This name will be rendered on NetWitness UI and REST as display name. -->
<!-- <description>User name to connect to eventsource</description> : Description will be displayed as
tool tip on NetWitness UI. -->
<!-- <value></value> : default value for parameter when no values are provided during configuration. -->
<!-- <clientconfigname>uname</clientconfigname> : Variable you would refer inside your python script to access this parameter value. -->
<!-- <clientconfig>true</clientconfig> : If set to true, this parameter will be pass down to your python script. -->
<!-- <required>yes</required> : Make parameter mandatory or optional. Value can be yes or no. -->
<!-- <validationType>RegularExpressionMatcher</validationType> : You can use validators to validate your parameter input. Refer table below to see supported validators. -->
<!-- <validationArgs>^[a-z0-9_-]{3,15}$</validationArgs> : Pass comma separated arguments to validator. -->
<!-- </parameter> -->
<parameters>
```

Use any of the following validators to validate input value for any parameter:

- BoolValidation
- UnsignedRangeValidation(lo,hi,bAllowZero)
- RegularExpressionValidation(str)
- RegularExpressionMatcher(str)
- NotEmptyValidation
- StringEnumValidation("str1,str2,...")
- SignedRangeValidation(lo,hi,bAllowZero)
- HostNameOrAddressValidation(bAllowEmpty)

This is a rendered view of the typespec on REST:

Password (password) (*)	DCC830F5331823EF4639AAB521E44:	Set
Polling Interval (polling_interval) (*)	180	Set
Start Date (start_date) (*)	85	Set
TypeSpec Version (typespec_version) (*)	1.0	
User Name (username) (*)	sa	Set

See the Type Spec XML File section under the API Reference for more details.

Exposing Configuration Parameters in the UI

You need a display properties file to render list of parameters on NetWitness UI, **<plugin>_ properties_en.xml** . If you want to support multiple internationalization locales for the UI displayed information, just duplicate the file and change the final 2 characters in name of the file from **en** to any other locale. Then open the file and modify display names for the desired language. Add all the parameters you have defined in the **<plugin>.xml** file to **<plugin>_ properties_en.xml** file.

Here is sample parameter block from properties file.

```
<field name="username" prettyName="User Name">
      <displayType>textfield</displayType>
      <mandatory>true</mandatory>
</field>
<field name="password" prettyName="Password" hidden="true">
      <displayType>textfield</displayType>
      <inputType>password</inputType>
      <emptyText>******</emptyText>
      <mandatory>true</mandatory>
</field>
```

```
<field name="start_date" prettyName="Start Date">
        <displayType>numberfield</displayType>
        <defaultValue>0</defaultValue>
        <mandatory>true</mandatory>
        <minValue>0</minValue>
        <maxValue>89</maxValue>
</field>
<field name="command_args" prettyName="Command Args" advanceField="true">
        <displayType>textfield</displayType>
</field>
```

Field attributes:

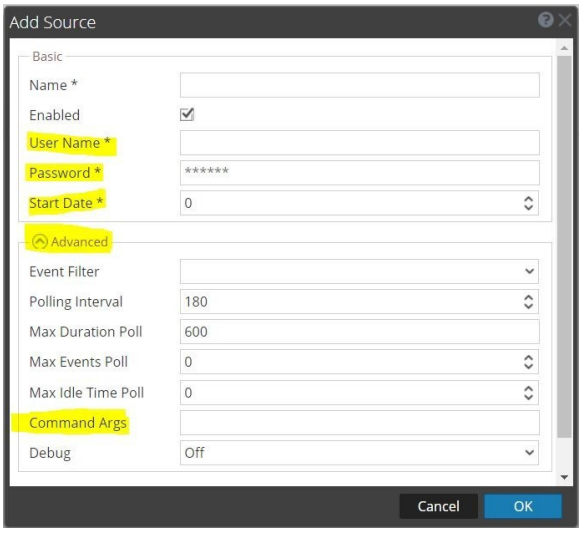
- **name**: field name, same as typespec file
- **prettyName**: - display name (label on UI)
- **hidden**: if set to true, values entered for that fields will be hidden
- **advanceField**: fields set as true will be available when you click on the Advanced section in the UI.

Inside the field, you can define the following tags:

Tag	Description
<description>	String description for the field
<displayType>	Used to render the parameter input on the UI Available options: • **textfield**: text input box, single line • **textfieldarea**: text input box, multiple lines • **numberfield**: numerical input box • **checkbox** • **combobox**: drop-down box
<mandatory>	True or False, indicates whether a value is required.
<defaultValue>	Set a default value for the field.
<minValue> and <maxValue>	Available for **numberfield**. When specified, performs range validation on the field.

Tag	Description
<validationRegex>	Available for **numberfield**. When specified, performs regex validation on the field.
<uncheckedValue> and <checked>	Available for **checkbox**. Sets Boolean true/false for checked or unchecked box.

This is a rendered view of the properties file in the UI:



See the Display Properties File section in the API Reference chapter for more details.

Defining CEF Parser Mapping

Plugins collect events into JSON format. The Log Collector converts these JSON events into CEF-formatted Syslog, then sends the Syslog data to the Log Decoder for parsing.

This JSON-to-Syslog transformation is performed by the Log Collector transformation code in /etc/netwitness/ng/logcollection/content/transform/cmdscript/***plugin_name***_transform.xml. This file contains the list of fields from which you need data. Generally, you select a subset of all the available fields, and only transform the ones you need.

The following code shows a sample JSON event collected by the new plugin.

```
<-- sample json event collected using new plugin -->
{
    "username": "Administrator",
    "httpRequest": {
        "clientIpAddress": "1.1.1.1",
        "method": "PUT"
    },
    "operationName": {
```

```
        "localizedValue": "Update group",
        "value": "Microsoft.Resources/subscriptions/resourcegroups/write"
      },
      "Severity": {
            "localizedValue": "medium",
            "value": ""
      }
}
```

The following code shows a sample transform file to transform the above event.

```
<!-- sample transform file to transform above event -->
<parameterTranslations>
<!-- Here is parameter translation for sample json log-->
<!-- <parameter> -->
<!-- <keyname>username</keyname> Variable name in json object, for nested
structure use .(dot) -->
<!-- <translatedName>uname</translatedName> rename keyname in CEF logs. -->
<!-- <valuemap></valuemap> provide valueMap you have defined to translet value for
 specified parameter. -->
<!-- <keep>true</keep> set to true to have this field as part of key=value pair in
CEF log. -->
<!-- <header></header> -->
<!-- </parameter> --> <!-- <parameter> -->
<!-- <keyname>Severity.localizedValue</keyname> -->
<!-- <translatedName>severity</translatedName> -->
<!-- <valuemap>severity</valuemap> Since value for severity is "medium" it will be
converted to 5 using defined valueMap on the top. -->
<!-- <keep>false</keep> Setting it to false will remove it from key value pairs,
This is required when you plan to put this field value in header. -->
<!-- <header></header> Set what header value you want to replace with this
parameter. Options: name,signatureId,severity.--> <!-- </parameter> --> <!--
<parameter> -->
<!-- <keyname>httpRequest.clientIpAddress</keyname> Example to demostrate how to
fetch nested values. -->
<!-- <translatedName>clientIp</translatedName> Renaming it to meaningful name. -->
<!-- <valuemap></valuemap> -->
<!-- <keep>true</keep> -->
<!-- <header></header> -->
<!-- </parameter> -->
</parameterTranslations>
```

The transformed CEF event:

```
Jul 27 2017 17:32:19 1.1.1.1 CEF:0|RSA|Netwitness Test Plugin Log
Collector|0|||5|uname=Administrator severity=medium clientIp=1.1.1.1
```

To parse transformed events, the Log Decoder uses the CEF parser. You need to update the CEF parser on the Log Decoder (`/etc/netwitness/ng/envision/etc/devices/cef/cef.xml`).

1. Add the new plugin event source entry in the CEF parser under the `<VendorProducts>` block. For example, `<Vendor2Device vendor="RSA" product="NetWitness Test Plugin Log Collector" device="testplugin" group="Cloud"/>` Events collected by the new plugin will appear as "testplugin" eventsource in the Investigator service.
2. The `<ExtensionKeys>` block defines mapping of "cefName" to "metaName".

   The `<translatedName>` in the transform file is the "cefName" in the CEF parser. So, if that variable does not exist, you need to add a new entry and map it to the correct meta variable.

   For example, from the above transform log, if "clientIP" does not exist in the CEF parser, we need to add this entry:

   ```
   <ExtensionKey cefName="clientIp" metaName="src.ip"/>
   ```

See the Transform XML File section in the API Reference chapter for more details.

Coding - Plugin Script

The majority of your coding time will be associated with connecting logic, event retrieval, and processing logic. The more time you have spent researching the event source and trying to connect and retrieve events from it, the easier it will be to transfer that knowledge and logic into the plugin framework.

The **<plugin_template>_collector.py** file contains some guidance on this work in the **on_collect** method.

```
# COLLECTOR-PLUGIN-CALLBACK 'on_collect'
def on_collect(self): """
```

```
This method is called by the BasePlugin parent class The derived plugin should...
```

```
1. Setup or initiate connection/session with event source
2. Retrieve one or more events
3. Process each event ...

     a) convert event data to TVM or JSON format
     b) Add book mark information to TVM or JSON format
     c) Publish event to Log Collector using self.publish_event
     (TVM/JSON)

4. Repeat steps 2-3 until

     a) there are no more events to collect
     b) An Abort is requested by Log Collector. This should be
     checked frequently enough in the processing loop to exit
     within 1-3 seconds maximum
     examples:
     if not self.is_aborted():

            ... process
```

```
      if self.is_aborted():
            ... exit processing loop
      while not self.is_aborted() and... :
            ... process
5. Exit on_collect method (allow for cleanup in parent class)
```

The following example code generates dummy events:

```
self.logger.info(self.get_plugin_type() + " COLLECTOR-PLUGIN-CALLBACK 'on_collect'
called")
self.logger.info("starting event processing loop, #events expected: %i",
self.args.number_events) self.logger.info("setting up connection here...")
# TODO: setup connection/session here...
# For this example, some dummy events are simply generated in a loop.
for i in range(0, self.args.number_events):
      magic_number = datetime.datetime.utcnow().toordinal()
      self.publish_event(
      {
            "severity": magic_number % 10,
            "type": "personal log" if i % 47 == 0 else "captain's log",
            "number": i,
            "starDate": magic_number / 10.0,
            }
      )
      is_aborted = self.is_aborted()
      if is_aborted:
            self.logger.warning("processing was aborted.")
            return
      time.sleep(0.05)
# Do some cleanup here, if necessary (for sessions, files, connections, etc.)
self.logger.info("finished event processing loop")
return
# TODO: ADD COLLECTION SPECIFIC METHODS
# ....
```

The following sections provide more guidance on coding the plugin.

Adding Connection Logic

For most eventsource types, there will be some kind of connection or session-starting logic that needs to occur to communicate with the eventsource. In many cases, there will be an API that is provided to interact with the eventsource. Hopefully, this API is available in python or is accessible through python.

Depending on how involved this is, you may add the logic directly in the COLLECTOR-PLUGIN-CALLBACK on**_collect** method or create one or two methods to handle the connection and session logic.

The complexity of writing the connection or session logic can vary greatly. Here are a few things to keep in mind:

- Start with the simplest connection logic required and get that working first. Add to the logic and re-work the logic as needed.
- Keep the connection as secure as the eventsource allows.
- If the connection is likely to fail, add a few retry attempts before exiting the plugin for the next polling interval run.
- Look at the existing plugins for examples. There maybe one with something similar that you can use as a model. However, do not assume similar event sources have similar logic. For example AWS CloudTrail and Microsoft Azure are both cloud based event sources, but they have very dissimilar APIs.

Adding Event Retrieval Logic

For most eventsource types, the event retrieval is done using the same API that is used for connecting to the eventsource. Depending on how involved this is, you may add the logic directly in the COLLECTOR-PLUGIN-CALLBACK on**_collect** method or create one or two methods to handle the event retrieval logic.

The complexity of writing the event retrieval logic can vary greatly. Here are a few things to keep in mind:

- Start with the simplest event retrieval logic required and get that working first. Add to the logic and re-work the logic as needed.
- Event chronological time order is very important. You want to retrieve the oldest events first and send them to the Log Collector in sequential order. Most event sources should provide the data in time order but not all (for example Azure).
- Keeping track of where you left off in the events is required so that you do not process the same events more than once. The plugin only needs to identify the bookmarking values and expose them as part of the payload sent to the Log Collector. See Adding Bookmarking Data for more details.
- The plugin's processing time is controlled through configuration values. In general, the plugin only has a few minutes to collect events before it is requested to stop and idle, so that other configured plugins and event sources can be processsed. This means you need to plan how many events you handle per iteration. It is best to keep the value within the 1000's of events

and iterate multiple times if more processing time is allowed for your plugin. Many APIs limit how many events can be retrieved in a single API call as well.

- If your plugin needs to run longer, this can be configured by the user. This may be necessary if you are processing a lot of data: but be aware that you are sharing processing time with other event sources.
- Look at the existing plugins for examples. There maybe one with something similar that you can use as a model.

Adding Event Processing Logic

The preferred format to send events to the Log Collector is using a JSON-encoded structure. It is also possible to use a Tag-Value-Map (TVM) structure, details of which can be found in the Event Payload Formats - Tag Value Map Event Format section.

This guide uses the JSON-encoded structure, shown here.

```
<EVENT format=json version=1 size=1001>>{"eventRaw": { event }, "bookmarkMeta": {
bookmark }, "collect ionMeta": { meta }}<<EVENT>
```

For this section we are interested in the **event** portion of the message. For many event sources, the events may already be in JSON format, which makes your work much easier. For others, you will need to translate the **event** portion into a JSON format.

Refer to Event Payload Formats - JSON Event Format and Event Processing Example (JSON) in the API Reference chapter for details.

Adding Bookmarking Data

Each event sent to the Log Collector contains the current bookmark information for that particular event. The Log Collector saves this information every time an event is handed off and published onto the RabbitMQ messagebus where ownership of the event is taken over by RabbitMQ. When the collection is stopped, the bookmark information is persisted by the Log Collector on the file system. Whenever the Log Collector re-invokes the plugin script, it passes this bookmarking data to the script as part of the configuration data.

The bookmark metadata consists of a set of name-value pairs that are configurable for each **cmdscript** event source type. These name-value pairs describe how to call the script next time it is called so it continues where it left off last time it was called.

This is extremely important, as it ensures that no events are skipped nor are any events processed more than once. Picking the correct bookmarking keys makes all the difference. In general, try to use as few keys as possible. It is also acceptable to include additional name-value pairs that you wish to persist between script processing runs: but keep these to a minimum, as it adds to the amount of data sent to the Log Collector for every event.

Publishing Events to Log Collector

Publishing of events to the log collector is the same for all plugins. The preferred format to send events to the Log Collector is using a JSON-encoded structure. It is also possible to use a Tag-Value-Map (TVM) structure, details of which can be found in the Event Payload Formats - Tag Value Map Event Format section.

The **<plugin_template>_collector.py** file contains some guidance on publishing events in the **publish_event(self, log_entry)** method.

```
def publish_event(self, log_entry):
```

```
entries = {}
entries['collectionMeta'] = {}
entries['collectionMeta']['logType'] = "event"
```

```
# the following meta entries specify 'bookmark' information,
# which help the Log Collector to resume processing without
# losing events or generating duplicate events:
```

```
entries['bookmarkMeta'] = {}
entries['bookmarkMeta']['lastModified'] = self.dt2str(datetime.datetime.utcnow(),
use_microseconds=True)
entries['bookmarkMeta']['recordNum'] = log_entry.get("number")
```

```
# NOTE:
# The 'eventRaw' entry is expected to contain the whole event in json format
(encoded as a string. This means that no 'contentMeta' entries are required. Most
cloud-based protocols return events in json format. Otherwise, you have to convert
'log_entry' into json first.
```

```
# Make sure to encode any special characters here, if necessary (in case they are
not UTF-8 compatible etc.): entries['eventRaw'] = json.dumps(log_entry, ensure_
ascii=False)
```

```
message = json.dumps(entries)
head = "<" + self.EVENT_STR + " format=json version=1 size=%i>>" % len(message)
tail = "<<" + self.EVENT_STR + ">"
event = (head + message + tail)
```

```
self.sync_print(event)
self.stats.inc_events(1)
```

The complexity of publishing is not too difficult, as this process is similar in all plugins.

Adding Logging Output

Adding additional logging information is a good idea, as all logging data is logged through the Log Collector as if it were an internal process. There is no need to create or manage log files and so forth. You do, however, need to be aware that too much logging cam create too much "noise" in the Log Collector logs. If some information is for debugging, use a debug level log so that it is only added to the log data if the event source has debug enabled.

The **<plugin_template>_collector.py** file contains many samples of logging, such as the following:

- `self.logger.debug("The phones are working again!")`
- `self.logger.info("received config!")`

  `self.logger.warning("received empty config!")`
- `self.logger.error("error parsing 'startDate' given in 'init' section '%s', using '%i' by default.", start_date_str, start_date_window_days)`

Note the following:

- There is no need to add context information like the time, data, eventsource name and so on, as that information is added automatically by the plugin framework.
- Messages logged at the **error** level appear as **FAILURE** in the Log Collector/NetWitness UI.

Adding Command Line Options

Additional command line options can be added to allow running the script from the command line for debugging. The **<plugin_template>_collector.py** file contains this section for adding additional command line options.

```
# COLLECTOR-PLUGIN-CALLBACK 'on_add_command_line_args'
def on_add_command_line_args(self, parser):
"""
    This method is called by the "BasePlugin" parent class on initial
    startup prior to parsing the command line arguments. This callback is
    used to add definitions of arguments which may be needed by this
    collection type. Normally these are only added so that the plugin
    script can be tested from the command line outside of the Log
    Collector scope.
"""
# TODO: ...
parser.add_argument("-n", "--number_events", action="store", type=int, default=50,
help="Number of generated dummy events.")
```

The above example specifies adding a command line option called **--number_events** that would be used to control the number of events to return before the script exits. In your logic, you can access the value of the option using **self.args.number_events**.

During testing, you normally run your plugin from the command line, redirecting in a JSON representation of the configuration values. This is equivalent to what the Log Collector does when starting the script. Another option is to expose some or all of the configuration values using command line options so that it is easier to debug your plugin outside of the Log Collector processing. A good example of this can be found in the **cloudtrail_collector.py** file associated with the CloudTrail AWS plugin.

Adding Configuration Initialization

Additional configuration initialization can be done to setup internal variables and other structures needed. The **<plugin_template>_collector.py** file contains this section for adding additional configuration initialization options.

```
# COLLECTOR-PLUGIN-CALLBACK 'on_config_init'

def on_config_init(self, config, bookmark):

    """
    :param config: A python dictionary of the configuration values
    (string,string) passed down from the Log Collector
    :param bookmark: A python dictionary of the persisted bookmark values
    (string,string) passed down from the Log Collector

    This method is called by the "BasePlugin" parent class on initial
    startup after receiving configuration data from Log Collector.

    The BasePlugin parent class stores Collection configuration items
    which can be retrieved using get_config_value(config_name)
    """

self.logger.info(self.get_plugin_type() + " COLLECTOR-PLUGIN-CALLBACK 'on_config_
init' called")
# TODO: ...
self.logger.debug("The phones are working again!")
if config is None:

    self.logger.warning("received empty config!")

else:

    # NOTE:
    # It's usually not a good idea to log the actual config here,
    # because it may contain security sensitive information!
    self.logger.info("received config!")

if bookmark is None:

    self.logger.warning("received empty bookmark!")

else:

    self.logger.info("received bookmark: %s", bookmark)
```

Note: No connection or event processing logic should be done in the **on_config_init** callback.

As an example, you could use command line options if the configuration is empty.

```
if config is None:

    self.logger.warning("No 'init' section found in CONFIG! Defaulting to
    command line parameters.")
    self.account_id = self.args.accountid
    self.region = self.args.region
    self.log_file_prefix = self.args.log_file_prefix
    self.bucket_name = self.args.bucket
```

```
        self.stats_interval = base_plugin.Stats.default_interval
        self.raw_event_encoding = None if (self.args.raw_event_encoding ==
        "none") else self.args.raw_event_encoding
        :
        :
else:
        self.account_id = init.get('accountId', self.args.accountid)
        self.region = init.get('region', self.args.region) # defaults to 'us-
        east-1'
        self.log_file_prefix = init.get('logFilePrefix', self.args.log_file_
        prefix)
        self.bucket_name = init.get('S3BucketName', self.args.bucket)
        self.region_endpoint = init.get('region_endpoint', self.args.region_
        endpoint)
        :
        :
```

Note: The command line options are only used when the script is run manually from the command console. All configuration values must be sent from the Log Collector process through the JSON configuration structure.

See Configuration Format in the API Reference chapter for more details.

Adding Configuration Change Support

Additional logic can be added to allow the script to receive changes to configuration values while it is running. The debug configuration value is automatically handled by the framework so any conditional logic using that value will automatically see the debug configuration value change. The majority of plugins will never run longer than a few minutes in a single execution, so this additional logic is rarely required.

The **<plugin_template>_collector.py** file contains this section for handling configuration changes.

```
# COLLECTOR-PLUGIN-CALLBACK 'on_config_change'
def on_config_change(self, config_change):
        :param config_change: A python dictionary of the configuration values
        that have changed (string,string) passed down from the Log Collector
        This method is called by the BasePlugin parent class when ever a
        configuration change is
passed down from the Log Collector. Only changed values will be contained in the
config_change dictionary.
"""
self.logger.info(self.get_plugin_type() + " COLLECTOR-PLUGIN-CALLBACK 'on_config_
change' called")
# TODO: ...
if config_change:
```

```
        self.logger.info("received config! change: %s", config_change)
else:
        self.logger.warning("no config change found")
```

Coding - Test Connection Script

The test connection script should use the same structure for most plugins. It is invoked when the **Test Connection** button in the UI is clicked.

It makes use of a slightly modified logger to keep track of warnings, errors and failures that might be created by steps during the connection / retrieval logic. Instead of logging log messages directly to **stdout**, it stores them in lists that can be used by the **report(self)** function to make sure there were no "failures" that occurred up to the point where the report function has been called.

In most cases, it is sufficient to modify the **run(self)** function shown below.

```
def run(self):
```

```
self.ptc.setup_argparser()
self.ptc.logger.info("starting %s version %s..." % (self.ptc.get_plugin_type(),
self.ptc.get_plugin_version())) self.check_config()
# initialize and start statistics thread
self.ptc.setup_stats_thread()
```

```
# Here we would check for things like:
# - authentication / user credentials
# - connecting to a cloud service correctly
# - potentially testing if downloading files is possible
# - etc.
self.check_authentication()
self.ptc.logger.info("finished run.")
```

```
# The report call gathers any logging calls that have been made to the
"ErrorTestLogger"
# and decides based on the presence of any "FAILURE" type log entries if the
connection was
# successful or not:
self.report()
```

This function calls additional functions that check specific steps of the collection plugin's sequence of establishing a connection to the event source and retrieving events from it.

For this purpose, it is very useful to instantiate the actual collection plugin and use its functions for those steps; sometimes in a simplified form, such as using dummy time ranges or default parameters.

Adding Connection Logic

Typically, there might be several steps to check for in the **run(self)** function. For example:

- parse the configuration received via **stdin**
- authenticate with the event source's API
- check if making a request to the event source succeeds without collecting events

We recommend that each of these steps should be using the actual functions from the corresponding collection plugin that is to be tested.

For example, if there is an **authenticate** function in the plugin that handles authentication, it could be called via a **check_authenticate(self)** function. In this "check"-function, you then could call the actual function with the configuration obtained in a previous step and surround this code block with a **try- except** block to catch any errors. Make sure to log meaningful error and failure messages for errors as specifically as possible. For example, avoid error messages such as "connection failed." Rather, construct more specific messages, such as "403 non-authorized error occurred while attempting to access..." or "network timeout error occurred: check connection or firewall."

The **run(self)** function must include a call to **self.report()** in order to verify that no failures have been encountered.

Adding Test Retrieval Logic

For the event retrieval logic, it can make sense to test certain preconditions that have to be fulfilled for the event source logs. For example, some APIs limit the time range for which events can be retrieved, or only allow to collect from certain sub-systems, groups, resources and so on.

Testing that it is actually possible to collect events can be somewhat tricky: at this point, we don't want to actually retrieve and collect events that are sent to the Log Collector. So, in many cases, we only want to see if the plugin function that makes the call to retrieve events does not result in any obvious errors. Thus, you should check for items such as malformed URLs, incorrect base-URLs, and insufficient permissions.

Verification and Troubleshooting

This chapter covers verification and provides some troubleshooting tips.

Verification

Verify the following behaviors:

- Verify script exits within 5 seconds when requested to stop by LC
- Verify script correctly handles being terminated by LC
- Verify events are in increasing chronological order
- Verify events parse correctly
- Verify events are bookmarked correctly

Verify script exits within 5 seconds when requested to stop by LC

The plugin script will be requested to stop by the Log Collector service whenever:

- the Log Collector is requested to stop
- the associated eventsource is changed to disabled by user
- the CmdScript collection type is requested to stop

This can all be handled by having your plugin logic check for an abort request from the Log Collector service. You can monitor for this condition in your code by using the following:

```
if self.is_aborted():
    ... exit processing loop
```

Verify script correctly handles being terminated by LC

If your script does not exit in time when given a stop request, the Log Collector process will terminate it. The delay time will depend on the type of stop being done. For a hardstop, such as a Log Collector process stopping, the delay time is 5 seconds. For a softstop, such as an event source disable, the delay time is normally slightly little longer.

Because of this, your script needs to be able to handle these types of stops. Normally, this is not an issue as the Log Collector is persisting the bookmark state for your script.

Verify events are in increasing chronological order

It is important to verify that the events are being consumed by the script in increasing chronological order. If the event source does not support this, you need to add logic to handle this. Much of the upstream processing and book marking relies on chronological order.

Verify events parse correctly

It is important that events are properly parsed and produce meaningful data. This can be a difficult task. You must check that parsing is not only correct, but also produces results that are meaningful when used for investigation. The data also needs to be mapped to existing meta keys whenever possible.

Verify events are bookmarked correctly

It is important to verify that the bookmarking data for each event is sufficient so that processing can resume from the last event collected after a restart.

Troubleshooting

The following items are things you can check when your plugin is not performing as expected.

- Run test script on command line to make sure connection logic is working.
- Run collector script on command line and verify events are coming in as JSON objects.
- If events are getting collected while running from the command line, then configure the event source using REST and run collection.
- Check stats on Log collector REST:

  `http://<LC_IP>:50101/logcollection/cmdscript/stats/eventsources/<plugin_name>/<plugin_eventsource>`
- Look for "Processing Error Count (errors)". If the error count is non-zero, check "last_execution_failures (*)" for latest error. For more information, enable debug and refer to logcollector logs (`tail -f /var/log/messages`).
- If logs are not coming to NwNetCat, check rabbitmq for any messages coming in for cmdscript and going out for transformation. Also check logcollector logs for any transform related errors.
- If the received syslog data on NwNetCat looks good, then start Log Decoder again and add that Log Decoder to Concentrator. Go to investigation and look for the new plugin event source.
- If events are coming up as unknown from your new plugin, make sure a vendor entry is added to the **cef.xml** parser on the Log Decoder.
- If any specific meta is not showing up, check your **cef.xml** parser to make sure transformed event variables are mapped to expected meta values.

API Reference

Using the REST API

Creating the Event Source Type

To configure plugin event source instances, you must first define the event source type.

1. Navigate to **http://host:50101/logcollection/cmdscript**.
2. Select the add property on the eventsources node.
3. Set the "eventsource_type" field to *plugin-type*, for example cloudtrail, azureaudit, and so on.
4. Send the request, and note the expected result ("Success").

Creating the Event Source Instance

Once the event source type has been added, one can now add event source instances.

1. Navigate to **http://host:50101/logcollection/cmdscript/eventsources**.
2. Select the add property on the *plugin-type* node.
3. Set the "name" field to the desired name.
4. Send the request, and note the expected result ("Success")
5. Navigate to **http://host:50101/logcollection/cmdscript/eventsources/plugin-type**.
6. Select the new event source instance.
7. Set the required and optional properties for the specific *plugin-type*.

Performing the Test Connection (Type Node)

1. Navigate to **http://host:50101/logcollection/cmdscript/eventsources**.
2. Select the test property on the *plugin-type* node.
3. Enter the required parameters.
4. Send the request, and note the result, hopefully ("Success").

Performing the Test Connection (Instance Node)

Assume there is a *plugin-type* event source instance with the name of "ONE".

1. Navigate to **http://host:50101/logcollection/cmdscript/eventsources/ONE**.
2. Select the test property on the ONE node.
3. Send the request, and note the result, hopefully ("Success").

Start or Stop Plugin Collection

Plugin collection is started using the **start** property, and stopped using the **stop** property on the logcollection/cmdscript node. Plugin collection may be automatically started via the **begin_collection_on_startup** field of the logcollection/cmdscript node. Automatic start of Command Script collection is disabled, by default.

Note: The start and stop operations apply to all plugin types and plugin instances. To avoid stopping all plugin eventsources, RSA recommends that you use the enable and disable operations on each plugin instance to control individual items.

Configuration Fields (Common)

All Plugin collection event source types share the same set of common configuration parameters common to all log collection event source types.

Name	Required	Default	Description
debug	no	false	Turns verbose NetWitness logging on.
enabled	no	true	Setting to false disables the collection from this event source.
event_filter	no	empty text	The name of the event filter object that is used to filter events.

Name	Required	Default	Description
event_filter_debug	no	0	Enables event filter debug. Set value by adding the options you want: • 0=Off • 1=Log Drops • 2=Log Accepts • 4=Log Non-Matching Conditions • 8=Log EventProtobuf. Examples: • To enable logging of dropped events and show matching and non-matching conditions set the value to 5 (1+4) • To enable full logging including dump of event protobuf contents set the value to 15 (1+2+4+8). Requires "Debug"
max_duration_per_cycle	no	120	Maximum duration, in seconds, of a polling cycle. A zero value indicates no limit
max_events_per_cycle	no	5000	Maximum number of events to pull from the event source during one polling cycle. A zero value indicates no limit.
max_idle_time_per_cycle	no	0	Maximum idle time, in seconds, of a polling cycle. A zero value indicates no limit.
polling_interval	no	180	Interval, in seconds, at which the event source should be polled for events. A value of -1 allocates an additional processing thread for event sources which support and are configured to run continuously
command_args	no	empty	Arguments to be added to the script when it is invoked. Use for extra arguments not in the normal configuration.

Plugin Specific Configuration Parameters

Each specific Plugin event source type will define additional configuration parameters that will be created specifically for that plugin event source type.

Command Script Protocol

This section defines the protocol used to communicate from the Plugin Collection and the Plugin script. Each message is contained in a tagged sequence similar to xml encoding. Refer to the "ChildProcess WorkUnit Specialization" section under the "Work Manager" chapter for more information.

Child Script listens for these items on STDIN:

- `<STOP>>`

  Sent from the workunit to the child process indicating for it to stop

- `<CONFIG>>{"init":{"configKey1": "configValue1","configKey2": "configValue2", .... "uploadDirectory": ".", "debug": 2}, "bookmark":{"persistenceKey1": "persistenceValue1", "persistenceKey2": "persistenceValue2", ... }, "change": {}}<<CONFIG>`

  Sent from the workunit to the child process indicating configuration data is new or changed.

Child Script produces these items on STDOUT:

- `<DONE>>`

  Indicates child process has completed allocated work and/or giving back control to workunit

- `<HEARTBEAT>>`

  Heartbeat sent from the child process to indicate it is still alive.

- `<EVENT>>`*`event data`*`<<EVENT>`

  `<EVENT format=xxx version=1 size=n>>`*`event data`*`<<EVENT>`

  Event envelope sent from the child process containing information associated with a single event. Collection meta keys are preceded by "*" in the name. Example:

  ```
  <EVENT format=tvm version=1
  size=95>>*PositionRecord=value,name1=value1,name2="value with , comma in
  side",name3=value3,nameN=valueN<<EVENT>
  ```

- `<STAT>>`*`statistics data`*`<<STAT>`

- `<CONNECT_SUCCESS>>`

  Connection to endpoint was successful.

- `<CONNECT_FAILURE>>`*`error message`*`<<CONNECT_FAILURE>`

  Connection to endpoint failed with error message describing the reason for the failure.

- `<LOG_INFO>>`*log data*`<<LOG_INFO>`

  String data sent from the child process that is processed as a LOG_INFO log message.
- `<LOG_WARNING>>`*log data*`<<LOG_WARNING>`

  String data sent from the child process that is processed as a LOG_WARNING log message.
- `<LOG_FAILURE>>`*log data*`<<LOG_FAILURE>`

  String data sent from the child process that is processed as a LOG_FAILURE log message.
- `<LOG_DEBUG>>`*log data*`<<LOG_DEBUG>`

  String data sent from the child process that is processed as a LOG_DEBUG log message.

Type Spec XML File

This section describes the structure and content of the event source type specification file used by plugin types.

The type specification file contains the following sections:

- Name, Type and Miscellaneous
- Collection Meta
- Collection Executables
- Persistence
- Configuration Parameters

The type specification file for each command script event source type is stored in the following directory location:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/<plugin>.xml`

For example, for cloudtrail the path is as follows:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/cloudtrail.xml`

Note: The typespec file is managed as Live content.

Name, Type, and Miscellaneous Section

The typespec file contains the following:

```
<name>plugin_template</name> <!--provide new plugin name -->
<type>cmdscript</type> <!-- Do not change -->
<version>1.0</version> <!-- If you change existing typespec bump up version by .1
-->
<author>administrator</author> <!-- This field can have any name -->
<description>Plugin Template Collection specification for eventsource type
"cmdscript"</description> <!-- Description for new plugin -->
```

- The <name> field defines the name of the plugin eventsource type. This value must match several other file and directory names.
- The <type> field defines the collection protocol collection type and this must always be **cmdscript**.
- The <version> field defines the version of the eventsource specification. Increment this value each time you make changes to the content of the plugin.
- The <description> field defines the description of the plugin eventsource type.

If you use the python script framework as described in Create Plugin Files using a Template (Python Script Framework), these items are handled automatically for you.

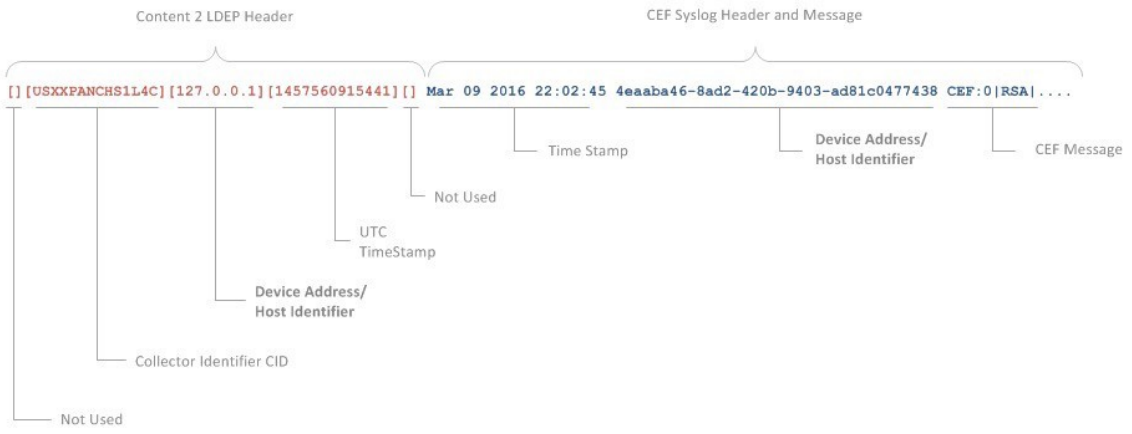
Collection Meta Section

The <collectionmeta> section contains the following:

```
<!-- Meta values collected in json logs will be used to populated host and
deviceaddress -->
<collectionmeta>
    <!-- MUST EDIT THIS -->
    <host></host>
    <deviceaddress></deviceaddress>
</collectionmeta>
```

- The <host> field defines the event's collection meta field name. This is used to identify the event source's host.
- The <deviceaddress> field defines the event's collection meta field name. This is used to identify the event source's device address.

Layout of Content 2 CEF Event as passed to Log Decoder:



Note that the same value or a different value can be used to identify the host in either header section. In the case of Azure Audit collection, two values are used: the <host> value identifies the connection source of the audit data, and the <deviceaddress> value adds context to the source of the message content as shown below.

```
<collectionmeta>
    <host>collectionMeta.subscriptionID</host>
    <deviceaddress>collectionMeta.tenantDomain</deviceaddress>
</collectionmeta>
```

Example CEF Event:

[][USXXPANCHS1L4C][127.0.0.1][1457560915441][] Mar 09 2016 22:02:45 4eaaba46-8ad2-420b-9403-ad81c0477438 CEF:0|RSA| Security Analytics Azure Log Collector|0|azure|azure|5| eventsataid=285261ec-c79f-49c8-9fff-ed8ceccaaf71resourceid= /subscriptions/4eaaba46-8ad2-420b-9403- ad81c0477438/resourcegroups/RSA-Azure-POC channels=Operation eventtimestamp= 2016-02-17T18:11:43.3008271Z submissiontimestamp=2016- 02-17T18:12:00.5741817Z subscriptionid=4eaaba46-8ad2-420b-9403-ad81c0477438 level=Informational caller=test.user@rsaglobaltest.onmicrosoft.com resourcegroupname=RSA-Azure-POC tenantid=d38362e1-3ba1-4efd-8772-a92abe105d92 operationid=6a443fc6-1c9a-40ea-a414-bd7422f0b1fc

Device section

The <device> section contains the following:

```
<device>
    <name>plugin_template</name> <!-- provide new plugin name -->
    <displayname>PlugIn Prettyname Log Collection</displayname> <!--
    provide display name, will be displayed as tooptip on NetWitness UI --
    >
</device>
```

The <device> section of a type specification is ignored by the log collector. It is intended to be used to add any device-specific information that may be useful to have. For example, the versions of the device that this specification supports.

Collection Section

The **<collection>** section contains information about how to invoke the plugin process or scripts. Normally, you only need to change the **plugin_template** text in this section to the name of your plugin.

If you use the python script framework as described in Create Plugin Files using a Template (Python Script Framework), these items are handled automatically for you.

Persistence Section

The **<persistence>** section contains information about the names of the variables of the event data or collection data that are to be stored as persistence variables. These are used to resume collection from where you left off. This is also referred to as bookmark data. See the Bookmarks and Persistence Files section, as well as Azure and CloudTrail implementations for examples.

The <persistence> section contains the following:

```
<!-- Define persistence variables, which you can use to resume collection from
where you left off. Below are sample examples. -->

<!-- You can add or remove more as per your script requirement. you can used same
name variable in your script to retrieve persisted values. -->

<persistence>

    <!-- MUST EDIT THIS -->

    <entry>

        <name>lastModified</name>
        <default></default>

    </entry>

    <!--<entry>-->

    <!--<name>line</name>-->

    <!--<default>0</default>-->

    <!--</entry>-->

    <entry>

        <name>recordNum</name>
        <default>0</default>

    </entry>

</persistence>
```

Parameters Section

The **<parameters>** section defines the additional configuration parameters or fields that are specific to the plugin type. The common configuration parameters listed in the Configuration Fields (Common) section are automatically included for all plugin types. This information specifically defines how the REST API configuration nodes are created on the Log Collector process when a plugin instance is created. The display of these configuration parameters is controlled by a separate **display_properties** configuration file. See the Display Properties File section for more details.

Great care should be taken to avoid changing or removing configuration parameters once a plugin type has been deployed. Once plugin instances have been created, migration maybe necessary to handle removal or change of parameter from version to version. The definition file version must also be incremented to indicate the change. For more details, see the Migration Section.

In the <parameters> block, you define a list of parameter you need for configuring the new plugin. The following validators are available to validate input values for parameters:

- BoolValidation
- UnsignedRangeValidation(lo,hi,bAllowZero)
- RegularExpressionValidation(str)
- RegularExpressionMatcher(str)
- NotEmptyValidation
- StringEnumValidation("str1,str2,...")
- SignedRangeValidation(lo,hi,bAllowZero)
- HostNameOrAddressValidation(bAllowEmpty) -->

Migration Section

The **<migration>** section contains information about how migration of configuration values is done from one version to the next. In general, changing or deleting configuration parameters is a bad idea and should be avoided.

You can, however, change the value of a configuration parameter. Doing this is relatively easy, using this section. Make sure that the **<migrate_if_version_less_than>** and the **<version>** values are appropriately set and incremented with a new version. If you do not need to make any changes, this section can remain empty.

Transform XML File

This section describes the structure and content of CEF Log Format Transformation definition file. The transform file for each plugin script event source type is stored in the following directory location:

```
/etc/netwitness/ng/logcollection/content/transform/cmdscript/<plugin>_
transform.xml
```

For example, for CloudTrail path is as follows:

```
/etc/netwitness/ng/logcollection/content/transform/cmdscript/cloudtrail_
transform.xml
```

This file defines how the collected event data is transformed into the CEF format log. You have the option of selecting the fields to be included, as well as transforming the name and values of the fields using transformations.

The configurable items in the transform include:

- Filter - includeUnknownParameters
- Filter - includeNullValueParameters
- Translation - escapeEqualSignsInValue
- Common Event Format (CEF)
- Translation Maps
- Global Translations
- Parameter Translations

Note: The transform file is managed as Live content.

CEF Section

The parameters in the **<commonEventFormat>** section are described in the following table.

Name	Description
host	A unique name for the event source sent by the client script
version	Identifies the version of the CEF format
deviceVendor	Strings that uniquely identify the vender of sending device
deviceProduct	Strings that uniquely identify the product of sending device
deviceVersion	Strings that uniquely identify the version of sending device
signatureId	Identifies the type of event reported
name	Represents a human-readable and understandable description of the event
severity	Reflects the importance of the event. Only numbers from 0 to 10 are allowed. 10 indicates the most important event. This value will be used as the default. It can be over-ridden with a severity specified in the events below.

Translation Maps Section

A translation map defines a set of named, string-to-string translations. These translations can be used to translate the value of a key/value pair parameter. For example, suppose there is a field that denotes the importance of an event. The problem is that the importance field is a string, not a number. We would like to use the importance field to drive the severity field in the CEF header. The severity field requires an integer between 0 and 10. So a translation map could be defined that would map the importance strings to a number between 0 and 10.

```
<translationMaps>
<!-- This map is used to translate a field value to a number which can then be
used as a severity -->

      <valueMap>

            <name>severity</name>
            <entry><string>very low</string> <value>0</value></entry>
            <entry><string>low</string> <value>2</value></entry>
            <entry><string>medium</string> <value>5</value></entry>
            <entry><string>high</string> <value>8</value></entry>
            <entry><string>very high</string><value>10</value></entry>

      </valueMap>
      <!-- Other maps can be defined -->
      <valueMap>

            <name>transactionTypes</name>
            <entry><string>0</string> <value>overnight</value></entry>
            <entry><string>1</string> <value>immediate</value></entry>
            <entry><string>2</string> <value>secure</value></entry>
            <entry><string>3</string> <value>foreign</value></entry>

      </valueMap>

</translationMaps>
```

Global Translations Section

These translations are used across all parameter translations.

<!-- These translations are used across all parameter translations -->

```
<globalTranslations>

      <prefix></prefix> <!-- string prepended to each value field -->

      <suffix></suffix> <!-- string appended to each value field -->

</globalTranslations>

The <globalTranslations>` section contains information about how to add a prefix
and/or suffix to each value field.

This can be useful if say

you need to add quotes around the value fields. In this case the prefix and suffix
values would be set to a double quote `"` for example
```

Parameter Translations Section

The parameters in the **<parameterTranslations>** section are described in the following table.

Key Word	Description
keyname	The name of the key as sent by the source
translated Name	The name to translate the key to
value map	An associative map that maps strings to strings
keep	Keep the parameter in the extension (**true** or **false**)
header	The CEF header field into which the value goes. Valid values include: • signatureId • name • severity
prefix	The string prepended to each value field
suffix	The string appended to each value field. The prefix and suffix will over-ride the global setting.

Sample code:

```
<parameterTranslations>
<parameter>
    <keyname>eventName</keyname>
    <translatedName>description</translatedName>
    <valuemap></valuemap>
    <keep>false</keep>
    <header>name</header>
</parameter>
<parameter>
    <keyname>eventType</keyname>
    <translatedName>type</translatedName>
    <valuemap></valuemap>
    <keep>false</keep>
    <header>signatureId</header>
</parameter>
:
:
</parameterTranslations>
```

Display Properties File

This section describes the structure and content of display properties specification file used by plugin types. The type specification file for each command script event source type is stored in the following location:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/<plugin>/<plugin>_properties_en.xml`

For example, for CloudTrail, the path is as follows:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/cloudtrail/cloudtrail_properties_en.xml`

The multiple display properties files can be created to support localization to other languages. The English en language type is required as the default. The selection of the properties file is based on the localization settings of the users UI environment. If a language type is not found, the English version is used.

Note: The display properties file is managed as Live content.

The **<sourceFields>** section of the display properties file defines how and which of the configuration parameters defined within the Typespec XML file <parameters> section are viewable in the Log Collector configuration UI.

```
<!-- This file is used to render typespec parameters on NetWitness UI -->
<?xml version="1.0" encoding="UTF-8"?>
<collectionTypes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="collection- source-config.xsd">
```

```
<!-- No changes required -->
<collectionType name="cmdscript" prettyName="cmdscript"
path="logcollection/cmdscript" version="10.6.2.0.0">
```

```
<!--Optional:-->
<description>string</description>
```

```
<!-- Replace name, prettyName and path with your new plugin name -->
<typeSpec name="plugin_template" prettyName="Plugin Name"
path="eventsources/plugin_template" version="10.6.2.0.0">
```

```
<!-- List fields you want to render on UI -->
<sourceFields>
<!--List all the field you have defined in plugin_template.xml typespec file-->
<!--Field name and prettyName are required -->
<!--Inside field tag you can define below tags -->
<!--
```

```
<description> - String description for field
<displayType> - This is used to render parameter input on UI.
Available options:
```

```
textfield - text input box, single line
textfieldarea - text input box, multiline
numberfield - number input box
```

```
            checkbox - check/uncheck box
            combobox - combobox drop down

        <mandatory> - True/false, indicates if a value is required.
        <defaultValue>- Set default value for field.
        <minValue> and <maxValue>

        - Available to numberfield, when specified it will do range validation
        on field.
        <validationRegex>
        - Available to textfield, when specified it will do regex validation
        on field.
        <uncheckedValue> and <checked>
        - Available to checkbox type, set bool true/false for checked or
        unchecked box.
        <inputType>

        <!-- Examples: these match the default items from the typespec. Remove
        if not needed -->

        <field name="username" prettyName="User Name">

            <displayType>textfield</displayType>
            <mandatory>true</mandatory>

        </field>
        <field name="password" prettyName="Password" hidden="true">

            <displayType>textfield</displayType>
            <inputType>password</inputType>
            <emptyText>******</emptyText>
            <mandatory>true</mandatory>

        </field>
        <field name="start_date" prettyName="Start Date">

            <displayType>numberfield</displayType>
            <defaultValue>0</defaultValue>
            <mandatory>true</mandatory>
            <minValue>0</minValue>
            <maxValue>89</maxValue>

        </field>

</sourceFields>
```

Program Script File

The program script for each plugin type is stored in the following directory location:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/<plugin>/<plugin>_collector.py`

For example, for CloudTrail, the path is as follows:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/cloudtrail/cloudtrail_collector.py`

Note: The display properties file is managed as Live content.

Associated Files and Required Libraries

There is a dedicated directory for each command script event source type. The location is:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/<plugin>`

For example, for CloudTrail, the path is as follows:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/cloudtrail`

This directory contains all of the required files to run the plugin script, as it is the root of the execution sandbox The **lib** sub-folder normally contains any required external libraries or resource files.

Test Connection Program Script File

A test connection script is provided with each plugin type to be used to test the connectivity to the event source. The test connection script for each plugin type is stored in the following directory:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/<plugin>_test_connection.py`

For example, for CloudTrail, the path is as follows:

`/etc/netwitness/ng/logcollection/content/collection/cmdscript/cloudtrail/cloudtrail_test_connection.py`

The test script is similar to Program Script execution, however, the following rules apply:

- The initialization JSON object is still sent to the script.
- The script uses the parameters from the initialization object to make a connection.
- If the connection succeeds, the script should respond with <CONNECT_SUCCESS>>.
- If the connection fails, the script should respond with <CONNECT_FAILURE>>*error_message*<<CONNECT_FAILURE>, where *error_message* is a message that explains the reason for the failure.
- Events should not be generate (do not send <EVENT>> tokens).

Bookmarks and Persistence Files

Each Plugin Collector is able to keep track of where the last successfully published event was created. This information is saved after each event, so that in case of an error publishing an event into the Log Collection framework, or a shutdown in the middle of processing, the Log Collector can pick up again starting at the last successfully published event and continue publishing from that point. Refer to the Persistence Section in the Type Spec XML File section for how to define the parameter values to store in the bookmark persistence files.

Persistence files:

- Are stored under the `/var/netwitness/logcollector/runtime/cmdscript/eventsources` folder.
- Are stored in XML format
- Are named by the eventsource type and directory name (<eventsource_type>.<directory_name>.xml), which are unique across all file collection event sources on a given node.
- For example, the persistence file for the CloudTrail instance "ONE" would be as follows:

```
/var/netwitness/logcollector/runtime/cmdscript/eventsources/cloudtrail.ONE.xml
```

Statistics (Common to all Plugins)

This section describes the aggregate statistics and event source statistics.

Aggregate Statistics	
Parameter	**Description**
Parameter	Description
total_errors	The number of processing errors by the collector since last restart.
total_errors_rate	The number of processing errors per second by the file collector since last update.
total_events	The number of events processed by the file collector since last restart.
total_events_rate	The number of events processed per second by the collector since last update.
total_filtered_events	The number of events filtered by the collector since last restart.
total_filtered_events_rate	The number of events filtered per second by the collector since last update.

Event Source Statistics	
Parameter	**Description**
errors	The number of processing errors by the eventsource since last restart.
events	The number of events processed by the event source since last restart, does not include filtered events.
events_rate	The number of events processed per second by the event source since last update, does not include filtered events (events per second).
filtered	The number of events filtered by the event source since last restart.
filtered_rate	The number of events filtered per second by the event source since last update (events per second).
last_ execution	Last successful work execution time
last_ execution_ duration	Last successful work execution duration
last_ execution_ failure	Last failed work execution time
state	The current state of the file collector.
total_ execution_ duration	Total successful work execution duration time since last restart.
total_ executions	Total successful work executions since last restart.

Logging

The Command Script collection module from the Log Collector generates various log messages:

- When collection is started or stopped
- When an event source is scheduled for collection
- Collection summary status when an event source collection ends

- When debug is enabled, debug messages from the Work Manager about the progress and result of the event source collection

The log level is controlled by the value of the debug setting for the event source. When it has a value of 0, no DEBUG messages are logged. When it has a value of 1 (verbose) or 2 (extended verbose), DEBUG messages are logged. This log level is also passed to the plugin as part of the event source configuration. The plugin can then enable DEBUG messages based on this value. Changes to this are also passed to the plugin script if it is already running. That allows changing the log level for long-running plugins.

The Command Script Protocol section details the log messages supported and their structure. The plugin must generate these messages at the appropriate levels (INFO, WARNING, FAILURE and DEBUG) and write them to **stdout**. The Command Script Work Unit captures these log messages and logs them to the standard NetWitness logging framework. All such messages are stored in a log database and are available in the UI for the Log Collector service. All except DEBUG messages also get logged to the `/var/log/messages` file via the **rsyslog** daemon. Note that these messages may be truncated per the **rsyslog** configuration.

The following are some log messages during normal operation from the `/var/log/messages` file. The messages with the tag **CloudTrailCollector** were generated by the plugin.

```
Jan 13 20:27:35 NWAPPLIANCE2598 NwLogCollector[12426]: [CmdScriptCollection]
[info] cmdscript, cmdscri pt started.

Jan 13 20:27:36 NWAPPLIANCE2598 NwLogCollector[12426]: [CmdScriptCollection]
[info] [cloudtrail.CT1] [ starting] Collection state not found, normal for first
collection attempt from an event source: /var/ne
twitness/logcollector/runtime//cmdscript/eventsources/cloudtrail.CT1.xml: cannot
open file

Jan 13 20:27:39 NWAPPLIANCE2598 NwLogCollector[12426]: [CmdScriptCollection]
[info] [cloudtrail.CT1] [ processing] [WorkUnit] [processing] 2017-01-13T20:27:39Z
CloudTrailCollector starting CloudTrailCollec tor version 1.0...

Jan 13 20:27:39 NWAPPLIANCE2598 NwLogCollector[12426]: [CmdScriptCollection]
[info] [cloudtrail.CT1] [ processing] [WorkUnit] [processing] 2017-01-13T20:27:39Z
CloudTrailCollector waiting for config on STD IN...

Jan 13 20:27:39 NWAPPLIANCE2598 NwLogCollector[12426]: [CmdScriptCollection]
[warning] [cloudtrail.CT1] [processing] [WorkUnit] [processing] 2017-01-
13T20:27:39Z CloudTrailCollector No 'lastModified' date time specified in
'bookmark' config section, using 'startDate' to determine initial window (in days)
for collecting logs from

Jan 13 20:28:08 NWAPPLIANCE2598 NwLogCollector[12426]: [Engine] [info] Child
process 12503 sent signal code: exited, child exit code: 0

Jan 13 20:30:09 NWAPPLIANCE2598 NwLogCollector[12426]: [CmdScriptCollection]
[info] cmdscript, cmdscript stopped.
```

Configuration Format

The configuration information and persistent bookmark information is passed to the child process just after invoking the child process. It is passed via standard input using JSON-based configuration enclosed in a <CONFIG>> tag in the **Command Script Protocol**.

```
<CONFIG>>{"init":{"configKey1": "configValue1","configKey2": "configValue2", ....
"uploadDirectory": ".", "debug": 2}, "bookmark":{"persistenceKey1":
"persistenceValue1", "persistenceKey2": "persistenceValue2", ... }, "change":
{}}<<CONFIG>
```

The <CONFIG>> JSON consists of 3 distinct subsections:

- **init**: contains the list of configuration parameters and values.
- **bookmark**: contains the list of persistence parameter values, from where collection should resume or start.
- **change**: normally empty, but can be passed at any time during child process execution to indicate that a configuration parameter value has changed. The debug flag is handled automatically when using the Python Script Framework.

Event Payload Formats

A command script outputs to STDOUT events in one of two formats:

- Tag Value Map, or
- JSON

Event Payload Formats - Tag Value Map Event Format

The Tag Value Map (TVM) event is structured as follows:

```
<EVENT format=tvm version=1 size=1129>>name1=value1,name2=value2, ...
,*name3=value3,*name4=value4 ...
<<EVENT>
```

- A name without the prepended * is mapped to content values in the generated event protobuf.
- A name with a prepended * is mapped to collection meta values in the generated event protobuf.
- The content and collection values can come in any order and can be mixed.

Event Payload Formats - JSON Event Format

The JSON event is structured as follows:

```
<EVENT format=json version=1 size=1001>>{"eventRaw": { event }, "bookmarkMeta": {
bookmark }, "collectionMeta": { meta }}<<EVENT>
```

Where ***event*** is of the following form:

```
{
    "eventVersion": "1.02",
    "eventID": "d8b308ff-1391-4fbc-9341-95e1762a0be4", "eventTime": "2014-
    12-01T15:51:16Z",
    "eventType": "AwsApiCall", "userIdentity":
    {
        "invokedBy": "support.amazonaws.com", "type": "Root",
        "principalId": "907171583576",
        "accountId": "907171583576"
    },
    "eventSource": "autoscaling.amazonaws.com"
}
```

Note that there can be nested name-value pairs. The value for `userIdentity` is a set of name-value pairs. Consider the name `principleId`. The name generated for it will be `userIdentity.principleId` There is no limit on the number of nested levels.

Where ***bookmarkMeta*** is of the following form:

```
"bookmarkMeta":
{
    "recordNum": "55",
    "lastModified": "2014-11-06T18:39:52Z",
    "line": "0"
}
```

The bookmark metadata consists of a set of name-value pairs that are configurable for each cmdscript event source type. These name-value pairs describe how to call the script each time, so that it always continues from the place where it left off.

Where ***collectionMeta*** is of the following form:

```
"collectionMeta":
{
    "logType": "event",
    "bucket": "sa-vlc-ct"
}
```

Collection meta is a name-value pair that can assigned to a field in the CEF header.

The above event would be parsed and put into the following protobuf format:

```
collection_meta:
    "lc.lpid" : "cmdscript.cloudtrail"
    "lc.cid" : "USENBENJERRYL3C"
    "lc.srcid" : "72.34.100.10"
```

```
"lc.ctype" : "cmdscript"
"lc.ctime" : "1366776771055"
"logType" : "event
```

```
content_meta:
```

```
"eventVersion": "1.02"
"eventID": "d8b308ff-1391-4fbc-9341-95e1762a0be4"
"eventTime": "2014-12-01T15:51:16Z"
"eventType": "AwsApiCall"
 "userIdentity.invokedBy": "support.amazonaws.com"
"userIdentity.type": "Root"
"userIdentity.principalId": "907171583576"
"userIdentity.accountId": "907171583576"
"eventSource": "autoscaling.amazonaws.com"
```

Note that the nested parameters from the JSON object are mapped to a linear naming scheme.

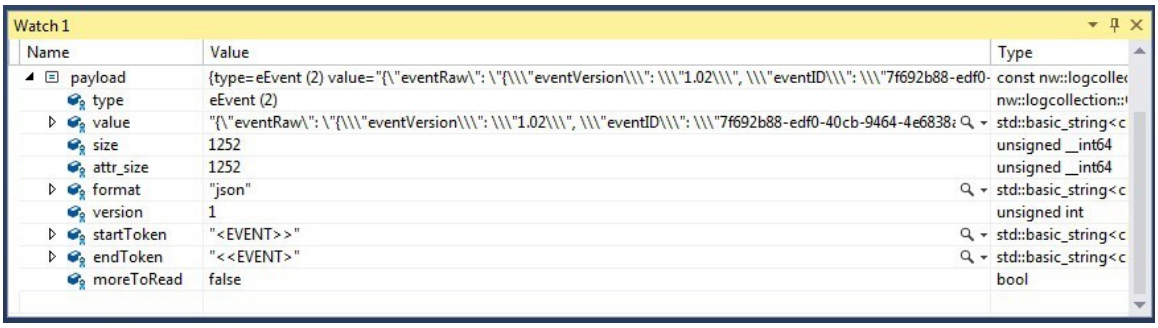
Event Processing Example (JSON)

The events are transmitted as JSON objects, where each JSON object contains a set of events. These objects are delineated with new lines, and the events are delineated by the JSON structure. The objects correspond to lines and the events correspond to records.

It is useful to see the progression of data transformations occurring in the work unit.

The Payload

The underlying child process work unit reads the messages from the script and calls the **onEvent** callback with the following structure:



The **value** field in the structure contains the payload that exists between the <EVENT>> and <<EVENT> tokens.

Structure and Content

The following is the JSON event extracted from `payload.value`. Note this is a string containing line breaks.

```
{"eventRaw": "{\"eventVersion\": \"1.02\", \"eventID\": \"7f692b88-edf0-40cb-
9464-4e6838a28598\", \"ev entTime\": \"2014-11-05T21:30:04Z\",
\"requestParameters\": {\"trailNameList\": []}, \"eventType\": \" AwsApiCall\",
```

```
\"responseElements\": null, \"awsRegion\": \"us-west-1\", \"eventName\":
\"DescribeTrail s\", \"userIdentity\": {\"principalId\": \"907171583576\",
\"accessKeyId\": \"ASIAIDKDOITHO5HMFCAA\", \"sessionContext\": {\"attributes\":
{\"creationDate\": \"2014-11-05T21:10:21Z\", \"mfaAuthenticated\": \"false\"}},
\"type\": \"Root\", \"arn\": \"arn:aws:iam::907171583576:root\", \"accountId\":
\"90717 1583576\"}, \"eventSource\": \"cloudtrail.amazonaws.com\", \"requestID\":
\"e7fcc4b8-6532-11e4-946c-05 5fef65b588\", \"userAgent\":
\"console.amazonaws.com\", \"sourceIPAddress\": \"168.159.213.209\", \"re
cipientAccountId\": \"907171583576\"}", "bookmarkMeta": {"recordNum": "0",
"lastModified": "2014-11-05 T21:34:50Z", "line": "0"}, "collectionMeta":
{"logType": "event", "bucket": "sa-vlc-ct", "S3Arn": "arn:aws:s3:::sa-vlc-
ct/cloudtrail-/AWSLogs/907171583576/CloudTrail/us-west-1/", "bucketKey":
"cloudtrail-/AWSLogs/907171583576/CloudTrail/us-west-1/2014/11/05/907171583576_
CloudTrail_us-west-1_20141105T2135Z_MfcG6rGY6HRfNO94.json.gz"}}
```


The **eventRaw** field contains the content for the event. Note the value of this field is parsed as a string, not a nested JSON structure. This requires we take the value of eventRaw, treat it as a JSON object and convert it to a ptree. This ptree is shown below. Note the raw event, bookmark meta, and collection meta sections.


```
{
    "eventRaw": "{
    \"eventVersion\": \"1.02\",
    \"eventID\": \"3c4b4864-d763-4c23-9cec-b58483bc667f\",
    \"eventTime\": \"2014-11-24T19:26:47Z\",
    \"requestParameters\": null,
    \"eventType\": \"AwsApiCall\",
    \"responseElements\": null,
    \"awsRegion\": \"us-east-1\",
    \"eventName\": \"GetAccountSummary\",
    \"userIdentity\": {
        \"invokedBy\": \"support.amazonaws.com\",
        \"type\": \"Root\",
        \"arn\": \"arn:aws:iam::907171583576:root\",
        \"principalId\": \"907171583576\",
        \"accountId\": \"907171583576\"
        },
    \"eventSource\": \"iam.amazonaws.com\",
    \"requestID\": \"d53be119-740f-11e4-a29d-6f23151a8ff4\",
    \"userAgent\": \"support.amazonaws.com\",
    \"sourceIPAddress\": \"support.amazonaws.com\",
    \"recipientAccountId\": \"907171583576\"
    }",
    "bookmarkMeta":
    {
        "recordNum": "0",
        "lastModified": "2014-11-24T19:29:45Z",
        "line": "0"
```

```
},
"collectionMeta":
{
    "logType": "event",
    "bucket": "sa-vlc-ct",
    "S3Arn": "arn:aws:s3:::sa-vlc-ct\/cloudtrail-
    \/AWSLogs\/907171583576\/CloudTrail\/us-west-1\/",
    "bucketKey": "cloudtrail-
    \/AWSLogs\/907171583576\/CloudTrail\/us-west-
    1\/2014\/11\/24\/9071715 83576_CloudTrail_us-west-1_
    20141124T1930Z_V24vfwXZwCio7CJb.json.gz"
}
}
```

Raw Event Extraction

The raw event section is extracted, as shown below, and converted to a set of **name:value** pairs which are inserted into the content meta of the protobuf event to be sent to the transform.

```
{
"eventVersion": "1.02",
"eventID": "3c4b4864-d763-4c23-9cec-b58483bc667f",
"eventTime": "2014-11-24T19:26:47Z",
"requestParameters": null,
"eventType": "AwsApiCall",
"responseElements": null,
"awsRegion": "us-east-1",
"eventName": "GetAccountSummary",
"userIdentity": {
    "invokedBy": "support.amazonaws.com",
    "type": "Root",
    "arn": "arn:aws:iam::907171583576:root",
    "principalId": "907171583576",
    "accountId": "907171583576"
},
"eventSource": "iam.amazonaws.com",
"requestID": "d53be119-740f-11e4-a29d-6f23151a8ff4",
"userAgent": "support.amazonaws.com",
"sourceIPAddress": "support.amazonaws.com",
"recipientAccountId": "907171583576"
}
```

Protobuf Values

The above ptree structure is recursively traversed to extract the name-value pairs for the event protobuf. Note the flattening of the parameter names using a dotted string format.

```
{
```

```
"eventVersion": "1.02",
"eventID": "3c4b4864-d763-4c23-9cec-b58483bc667f",
"eventTime": "2014-11-24T19:26:47Z",
"requestParameters": null,
"eventType": "AwsApiCall",
"responseElements": null,
"awsRegion": "us-east-1",
"eventName": "GetAccountSummary",
"userIdentity.invokedBy": "support.amazonaws.com",
"userIdentity.type": "Root",
"userIdentity.arn": "arn:aws:iam::907171583576:root",
"userIdentity.principalId": "907171583576",
"userIdentity.accountId": "907171583576",
"eventSource": "iam.amazonaws.com",
"requestID": "d53be119-740f-11e4-a29d-6f23151a8ff4",
"userAgent": "support.amazonaws.com",
"sourceIPAddress": "support.amazonaws.com",
"recipientAccountId": "907171583576"
}
```


Security Model

The CmdScript Plugin framework allows development of collection plugins by third party developers. Running such plugins on NetWitness appliances presents security challenges, since these third-party plugins can potentially access configuration and data files from the appliance. To control the privileges of these plugins, the plugin scripts are run within an SELinux Sandbox.

- During the RPM installation, the appliance is prepared to run SELinux in **Enforcing** mode. A custom SELinux policy is installed, compiled and loaded. If any of the configuration steps fail, the CmdScript collection protocol cannot be started, and the following error is returned:

  Collection cannot start due to configuration errors. Please run the command /opt/netwitness/bin/compile_load_plugin_module.sh again to remediate

  If following that suggestion on the Log Collector still does not resolve the problem, contact RSA NetWitness Log Collector Customer Support.

- Plugin scripts run as a non-privileged user. When a plugin is added to the Log Collector configuration, a user with the name **<plugin_name>_pl** is created. Plugin scripts for all event sources of that plugin type run as this user.
- Plugin scripts cannot download or write to any directory including `/tmp` area except into an assigned temporary directory. Each event source is assigned the following directory:

  `/var/netwitness/logcollector/scriptUpload/<plugin_name>/<eventsource_name>`

  These temporary directories are created when the plugin or event source is added.

- Required system files in `/etc` (for example `/etc/resolve.conf` for name resolution) will be readable by the plugin scripts. However, no files from `/etc/netwitness/ng` area (except the plugin content mentioned below) are readable.
- The plugin content area must have a specific SELinux context:

  `/etc/netwitness/ng/logcollection/content/collection/cmdscript/<plugin_name>`

  This context is set when the plugin is added to the Log Collector configuration. Some remediation is also undertaken automatically when the plugin scripts are run.

SELinux

The standard Discretionary Access Control (DAC) mechanism, based on user/group identity and ownership, does not ensure strong security. Security- Enhanced Linux (SELinux) is an implementation of a Mandatory Access Control (MAC) mechanism in the Linux kernel[1]. After checking the standard DAC, SELinux enforces rules on files and processes based on defined policies. Processes cannot access resources or transition to different roles or domains that are not explicitly allowed by a policy.

The Plugin Security model depends on SELinux. SELinux must be running in Enforcing mode. The Log Collector RPM installation scripts make the necessary changes so that Plugin Collection can be performed.

SELinux Sandbox

SELinux Sandbox is a utility that allows running untrusted programs while limiting access to system resources. SELinux sandbox supports running programs under various SELinux security domains. For example, the default **sandbox_t** domain prevents any network access.

Plugin collection requires pulling events from external sources, so SELinux Sandbox is used in **sandbox_net_client_t** domain which allows the plugin scripts network access.

SELinux ensures that sandbox and all child processes are restricted in the same way. So, malicious programs spawned within the sandbox cannot gain access to restricted resources.

Security Objectives

The default DAC settings on NetWitness systems do not prevent SELinux sandbox processes from accessing confidential configuration and data files. By default, all files under `/etc` have SELinux context settings that allows the sandbox to read them. The NetWitness services uses `/etc/netwitness/ng/` for their configuration. Some of the files under that directory need to be inaccessible to sandbox. This requires a custom policy to be created and security context of files in `/etc/netwitness/ng` to be changed.

Until 11.0, NetWitness appliances have not used SELinux in Enforcing mode. On 10.6.x, SELinux runs in Permissive mode that still checks the policy rules but does not enforce them. Changing SELinux to run in Enforcing mode impacts other NetWitness components such as Health & Wellness. Thus, the custom policy contains additional rules for granting needed access to those components.

The following are the objectives:

- Plugin scripts must run in the SELinux Sandbox as a non-privileged user
- Plugin scripts must have network access
- Plugin scripts must have read access to system files in `/etc`
- Plugin scripts must NOT have access to configuration files in `/etc/netwitness/ng` area (with below exception)
- Plugin scripts must have read/execute access to plugin content in `/etc/netwitness/ng/logcollection` area
- Plugin scripts must have write access only to assigned temporary directories
- Prevent plugin scripts from running any executable from assigned temporary directories. (Scripts can still be started.)
- There must not be a way to bypass the SELinux Sandbox from within the Log Collector.

Plugin Helper

The CmdScript Work Unit does not use the SELinux Sandbox directly. Instead, it invokes the Plugin Helper tool, **/opt/netwitness/bin/NwCmdscriptPluginHelper**. There is no configurable way to bypass the Plugin Helper. The Plugin Helper receives the plugin username, plugin name and script with arguments and performs the following steps:

- The Plugin Helper runs as the same user as the Log Collector (root) and remediates potential SELinux configuration issues by running the script **/opt/netwitness/bin/cmdscript_workunit_config.sh**. It checks and fixes missing user, ownership, and SELinux security context of Plugin content.
- Uses **setuid()** and **setgid()** system calls to switch to the specified non-privileged user.
- Executes the specified plugin script and arguments in SELinux sandbox.

Custom SELinux Policy

The SELinux sandbox typically can be used without any additional custom policy rules. However, in our case the sandbox is spawned by the Log Collector service. This requires security context transitioning from that of a service to a sandbox that was not covered by the sandbox's SELinux policy.

Additional rules cover the following:

- To prevent access to files in the `/etc/netwitness/ng` area, all files in that area (with the below exception) are typed as **netwitness_etc_t**. This is a new type created for the NetWitness services. The sandbox does not have access to this type.
- Plugin content files need to be accessible, so all files in the `/etc/netwitness/ng/logcollection/content/collection/cmdscript/` area are typed as **netwitness_lc_plugin_t**. This is a new type created for the plugin content files. the sandbox has read and execute access to this type.
- Plugin temporary directories in `/var/netwitness/logcollector/scriptUpload/` are typed as **sandbox_netclient_tmpfs_t**. This is a type available for temporary files and folders within the sandbox.
- Required accesses to **collectd**, **sshd** and **syslogd** services as well as additional accesses for sandbox are added.
- The policy files are installed in the `/etc/netwitness/etc/selinux` directory. The policy is compiled and loaded into SELinux during RPM installation.

Configuration Scripts

The following new scripts perform various configuration steps.

`/opt/netwitness/bin/compile_load_plugin_module.sh`

This script is run automatically during the RPM installation by the post-install script. It performs the following functions:

- Set SELinux mode to Enforcing mode in **/etc/selinux/config** so change is effective on reboot
- Change SELinux mode to Enforcing by running the command **setenforce 1**
- Compile the custom policy **/opt/netwitness/etc/selinux/logcollection_plugin.te** and reload
- Change SELinux context for files in the `/etc/netwitness/ng` area.
- On any error, create marker file **/etc/netwitness/logcollection/.SELinux_configuration_error_marker**. The CmdScript collection protocol cannot be started if this file exists. This script may be rerun to remediate the configuration errors.

`/opt/netwitness/bin/cmdscript_plugin_config.sh`

This script is run by Log Collector whenever a new plugin type (**eventsource_type**) is added to the CmdScript eventsources.

- Create user/group with **<plugin_name>_pl** name
- Create plugin temporary directory `/var/netwitness/logcollector/scriptUpload/<plugin_name>`
- Change ownership and group of the temporary directory
- Change the SELinux context for all plugin content in `/etc/netwitness/ng/logcollection/content/collection/cmdscript/`
- Change the SELinux context for all files and folders in the temporary directory

`/opt/netwitness/bin/cmdscript_plugin_cleanup.sh`

This script is run by Log Collector whenever a plugin type (eventsource_type) is deleted from the CmdScript eventsources.

- Delete **<plugin_name>_pl** user and group
- Delete plugin temporary directory `/var/netwitness/logcollector/scriptUpload/<plugin_name>`

`/opt/netwitness/bin/cmdscript_workunit_config.sh`

This script is run by the Plugin Helper tool, that is, whenever the CmdScript Work Unit initiates collection for a plugin event source. It remediates the CmdScript environment in case of changes since the initial configuration by the above scripts. It performs all the tasks from the **cmdscript_plugin_config.sh** script.

Troubleshooting SELinux Issues

SELinux related issues may impact the CmdScript plugins or other components installed on the appliance such as Health & Wellness (Collectd). The NetWitness services such as Log Decoder should not be impacted however. SELinux runs in targeted mode. Since no specific SELinux domain or policy is defined, they would run as unconfined services.

Access denied errors

If a plugin script attempts to access a resource that is not allowed, an "AVC" type "denied" log is generated in the **/var/log/audit/audit.log** file. Below is a sample log entry.

```
type=AVC msg=audit(1466791119.112:174784): avc: denied { transition } for pid=14457 com-
m="sandbox" path="/opt/rh/python27/root/usr/bin/python2.7" dev=dm-10 ino=27265 scon-
text=system_u:system_r:initrc_t :s0 tcontext=system_u:system_r:sandbox_net_client_t:s0
tclass=process
```

This indicates that the sandbox is not allowed to transition from the source context to the **sandbox_net_client_t** context.

Granting Access

If the denied error is something needed, it may be allowed by granting that access. The above audit log may be converted to a policy fragment allowing that access by using the **audit2allow** command.

```
# echo 'type=AVC msg=audit(1466791119.112:174784): avc: denied { transition } for pid=14457
comm="s andbox" path="/opt/rh/python27/root/usr/bin/python2.7" dev=dm-10 ino=27265
scontext=system_u:system_r: initrc_t:s0 tcontext=system_u:system_r:sandbox_net_client_t:s0
tclass=process' | audit2allow -r

require {

        type sandbox_net_client_t;
         type initrc_t; class process transition;

}

#============= initrc_t ==============

allow initrc_t sandbox_net_client_t:process transition;
```

The generated code can be merged into the custom policy. Once recompiled and reloaded, the test can be repeated to confirm that this error no longer occurs.

Checking SELinux Context of files and processes

You can display the security context of files and processes using the **-Z** option to the **ls** and **ps** commands.

```
# ls -lZ /etc/netwitness/ng/*.pem

-rw-r--r--. root root system_u:object_r:netwitness_etc_t:s0 /etc/netwitness/ng/appliance_cert.pem
-rw-r--r--. root root system_u:object_r:netwitness_etc_t:s0 /etc/netwitness/ng/appliance_dh2048.pem
-rw-r--r--. root root system_u:object_r:netwitness_etc_t:s0 /etc/netwitness/ng/logcollector_cert.pem
-rw-r--r--. root root system_u:object_r:netwitness_etc_t:s0 /etc/netwitness/ng/logcollector_dh2048.pem
-rw-r--r--. root root system_u:object_r:netwitness_etc_t:s0 /etc/netwitness/ng/logdecoder_cert.pem
-rw-r--r--. root root system_u:object_r:netwitness_etc_t:s0 /etc/netwitness/ng/logdecoder_dh2048.pem

# ps -efZ | grep NwLogCollector
system_u:system_r:init_t:s0 root 2067 1 4 Jan11 ? 01:11:59 /usr/sbin/NwLogCollector --
stopwhenready
```

Use the **restorecon** or **chcon** commands to change the security context of files

SELinux Sandbox References

See the following URLs for more information about the SELinux Sandbox.

- https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/chap-Security-Enhanced_Linux-Introduction.html

- http://serverfault.com/questions/655766/selinux-sandbox-fails-to-start-from-systemd-service/759622
- http://fedoraproject.org/wiki/PackagingDrafts/SELinux#Creating_new_types