Splunk® SOAR (On-premises)
Develop Apps for Splunk SOAR (On-premises) 5.4.0

Generated: 10/28/2022 9:30 pm

Table of Contents

Introduction

Splunk SOAR apps overview

Splunk SOAR apps provide a mechanism to extend Splunk SOAR (On-premises) by adding connectivity to third party security technologies in order to run actions. Given the broad set of technologies that can be orchestrated during a cyber response exercise, apps provide some relief in allowing users and partners to add their own custom functionality.

Splunk SOAR apps are developed by engineers knowledgeable in Python and modern web technologies.

To develop a Splunk SOAR app, start with the app wizard:

1. From the main menu, select **Apps**.
2. Click **App Wizard**.

Splunk SOAR app architecture

Splunk SOAR apps are written in Python to create a bridge between Splunk SOAR (On-premises) and other security device/applications. Think of them as having two strict edges:

- One of the edges is given an action to be carried out on behalf of Splunk SOAR (On-premises).
- An app on the opposite edge converts the action into specific commands to communicate with its device or service.

The result of these actions are read by the app and passed back to Splunk SOAR (On-premises). This simple design helps facilitate automated actions that are carried out by Splunk SOAR (On-premises) on behalf of the user.

The first edge is implemented by a rich set of Python APIs that the platform exposes to the app developer through a base class.

Apps distributed by Splunk SOAR or third parties are transmitted as `.gzip` archives that you can import into Splunk SOAR.

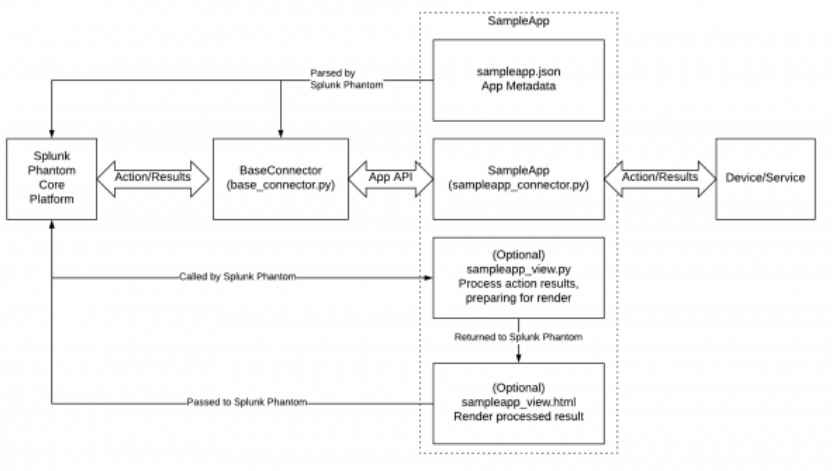
Splunk SOAR app components

A Splunk SOAR app consists of a number of components.

Component	Description
`__init__.py`	Required to initialize and define a Python package. You can use an empty file.
`sampleapp.json`	JSON metadata that describes the app and functionality that the app provides
`sampleapp_connector.py`	

Component	Description
	The App Main Connector Module (Python script) that implements the actions that are provided by the app. This module is a class that is derived from the BaseConnector class.
`sampleapp_view.py`	Optional widget view. This is a view, in the context of standard MVC framework. Splunk SOAR is built on Django, an open source Python-based MVC framework. Splunk SOAR will load views that you have specified within your JSON meta-data file dynamically. Full documentation on views and templates is available on the Django documentation website.
`sampleapp_view.html`	Optional widget template. The template defines how the information within the view is to be rendered and displayed. The full complement of Django tags are available within a template.

This image shows how the various components interact with each other.



App Wizard and Editor

Optional: Set up a development environment for Splunk SOAR (On-premises)

It is a good practice to develop your apps in a separate environment than your production Splunk SOAR (On-premises) deployment.

For instructions on how to install Splunk SOAR (On-premises), see How can Splunk SOAR (On-premises) be installed?.

Set up the dev environment

Once the virtual machine image file is downloaded, you must load it in your virtual machine software of choice, then set both the root and user passwords. Remote root login is disabled, so you need to use SSH as the user account the use sudo to get root access.

The firewall is pre-configured to allow SSH connections.

SSH to Splunk SOAR

```
localmachine:~ alice$ ssh user@192.0.2.255
user@192.0.2.255's password:
[user@localhost ~]$ sudo su -
[sudo] password for user:
[root@localhost ~]#
```

Set the hostname

Optionally, you can set the hostname of the virtual machine to make accessing it easier:

```
[root@localhost ~]# hostname appdev
[root@localhost ~]#
```

User 'phantom'

It's good practice to use a non-root user for app development. The user **phantom** is already created for your Splunk SOAR (On-premises) deployment and can be used for this purpose. You should set a password for this user account.

Set the password for the user **phantom**:

```
[root@localhost ~]# passwd phantom
Changing password for user phantom.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[root@localhost ~]#
```

Exit and login as **phantom**.

```
[root@localhost ~]# exit
logout
[user@localhost ~]$ exit
logout
```

```
Connection to 192.0.2.255 closed.
mymbp:~ alice$ ssh phantom@192.0.2.255
phantom@192.0.2.255's password:
[phantom@appdev ~]$
```

Create an app with the App Wizard

Use the App Wizard to develop custom security apps to use within playbooks, interface with APIs, or use external services.

The App Wizard is divided into multiple modules, fields, tabs, and files, each representing a logical part of an app. Most of the input elements within the App Wizard have an information icon. On mouse hover, these icons reveal detailed help tips, including naming conventions.

To create an app, complete the following steps:

1. Navigate to the **Main Menu**.
2. Select **Apps**.
3. Click **App Wizard**.

Enter basic app information

Define your app's characteristics by entering information in each of the required fields. The information you enter is used to generate the app's JSON file and fill in the **Basic Info** section in the app editor later.

1. Enter the basic app information for your app.
2. Click **Continue** to move to the editing portion of the app wizard.

The required information includes the app name, the app description, product vendor, product name, app publisher, the app's type, and icons in either SVG or PNG format for Splunk SOAR (On-premises)'s light and dark themes. To learn more about these characteristics see the topic Top level definition.

Any binary files you include in your app, such as the app's icon, are not viewable or editable in the App Wizard's editor.

Example app 'Get IP Info'

This example creates an app that uses the ipinfo.io service. The values given in this table are examples.

Field	Value
App Name	Get IP Info
App Description	`This app gets information from the <b>ipinfo.io</b> geolocation service.`
Product Vendor	ipinfo.io
Product Name	ipinfo.io
App Publisher	Splunk SOAR App Wizard example
App Type	information
App Logo for light theme	SVG or PNG image file
App Logo for dark theme	SVG or PNG image file

For an expanded list of an app's possible metadata, see Top level definition.

Add any dependencies

You have two options for managing any dependencies for your app, either by using wheel files or else PyPi to install packages. To add dependencies:

1. Click **+**.
2. Select either **wheel** or **pypi** from the dropdown menu.
    - ♦ select **pypi** to install packages, such as numpy, through Python's online package manager.
        - ◊ Enter the dependencies module name.
    - ♦ select **wheel** to install prepackaged resources through an uploadable file, for use in air-gapped systems or for customized needs.
        - ◊ Enter the dependencies module name.
        - ◊ Upload the dependencies wheel file. You can drag and drop the file into the space provided, or click the box to browse for the file.
3. Click **Confirm**.

For more information on specifying dependencies using pip, see Specifying pip dependencies.

Remove a dependency

If you added a dependency previously and have decided to remove it, follow these steps.

To remove a dependency:

1. Expand the **Dependency Details** section in the left-hand navigation panel.
2. Locate the dependency you wish to remove.
3. Click the trash can icon to delete the dependency.
4. Click **Confirm**.

Example app 'Get IP Info'

The example app does not have any dependencies.

Enter configuration details

You can use configuration details instead of hardcoded values for asset configuration. This is useful when the values for configuration options could change in some circumstances but you want to be able to control or limit the possible values.

1. Click **edit icon**.
2. Click **Add Configuration Option**.
3. Type a name for the configuration option. This name is used in the app's python code, and is not displayed in the user interface. Names should be all lowercase, using underscores to separate words. Do not use spaces or special characters.
4. Select a type for the option
    - ♦ string for a sequence of characters
    - ♦ boolean for a true or false
    - ♦ numeric for integers
    - ♦ password for secret identification
    - ♦ placeholder to specify the order that the controls are displayed to the user.
5. Add a description of the configuration option. This text is displayed in the user interface next to the configuration option.

6. If the configuration option must be set on the asset, check the box for **Required**.
7. If you have additional items to set, such as a default value, or you want to provide a list of preset values, click the **edit icon** in the **More** field.
    - If you need to set a default value for the asses, type it in the **Default** field.
    - If you want to provide a drop-down list of values to be used to the asset's configuration, type them in the **Value List** field.
8. Click **Update.**
9. To add more configuration options, click **Add Configuration Option**, and enter the details for that option.
10. When you have added all your configuration options, click **Confirm**.

See Configuration Section and Place Holder Data Type.

Example app 'Get IP Info'

Set a configuration option for using ipinfo.io.

The app needs to connect to https://ipinfo.io/<ip_to_query> in order to retrieve information about an IP address. The base URL for ipinfo.io, https://ipinfo.io could be hardcoded in the app, but for this example, add it as configuration option.

1. in the left hand navigation panel, in the **Configuration Details** section, click the edit icon.
2. Click the **Add Configuration Option** button.
3. Set the following values:

Field	Value
Name	base_url
Data Type	String
Description	The base URL to connect to retrieve IP information.
Required	Checked

4. Click **More**, set the default value to **https://ipinfo.io**, and click **Update**.
   Splunk SOAR will display this as the default value for users when they configure a new asset for this app.
5. Click **Confirm**.

Create app actions

You can select an action for the app to implement based on the type value you select for your app. The types of apps are devops, email, endpoint, identity management, information, network device, network security, reputation, sandbox, siem, ticketing, and virtualization.

1. Click **+**.
2. Select the type of app you're creating or editing from the **App Category** drop-down list.
3. Select the action you need from the the **Available Actions** drop-down list. Some examples follow:
    - lookup ip returns the resolved host name.
    - geolocate ip queries service for IP location information.
    - test connectivity validates the asset configuration for connectivity using supplied configuration.
4. Edit the parameters for the action if needed. For most included actions, you can simply use the parameters provided. For example, for the geolocate ip action all the parameters are usable as is.
5. When you have added the actions for your app, click **Save** to save your app.

You can also add a custom action if you want to customize your actions' parameters.

1. Click **+**.
2. Click **Switch to Custom Action**.
3. Enter values the following fields:

Field	Value
Action Name	Name of the action, in lowercase. Names may only contain alphanumeric, spaces, and underscores. Names should follow noun + verb conventions, like "list processes" or "detonate file".
Description	A very brief description of the action. A more detailed version can be included in the **Verbose Description** field later.
Action Type	Select from the menu provided.
Read Only	Check this box if the action does not change values or states of other items.
REST Endpoint	The REST API endpoint the action should use in conjunction with a base URL.
Verbose Description	Enter a more detailed explanation of the action and the expected results.

4. Click **Add Parameter** to enter any required values for your custom action:

Field	Value
Name	The parameter's name.
Data type	Select a type from the menu.
Description	This is a description that will be displayed next to the **Launch Action**dialog. It is not a name, but a description of the action the user is about to run. Be concise, and avoid long descriptions that will wrap lines.
Required	If this parameter is required to run the action, this box should be checked.
Primary	This value is used in conjunction with **Contains** to display a list of contextual actions. If this parameter is the primary from a range of options available, this box should be checked.
Contains	Specifies the sort of information the parameter contains. This allows for matching of one action's output parameter to another action's input parameter.

5. Click **Confirm**.
6. When you have added the actions for your app, click **Save** to save your app.

See Naming Actions for more information about naming your actions.

Example app 'Get IP Info'

The example app is type **information** and uses the **geolocate ip** and **lookup ip** actions. For this example use the default values.

1. In the left navigation, click the **+** icon in the **Actions** section.
2. Select **geolocate ip** from the **Available Actions** drop-down menu.
3. Set the **REST Endpoint** to /**geo**.
4. Click **Confirm**.
5. In the left navigation, click the **+** icon in the **Actions** section.
6. Select **lookup ip** from the **Available Actions** drop-down menu.
7. Set the **REST Endpoint** to /**hostname**.
8. Click **Confirm**.

Edit the **test connectivity** action.

1. Expand the **test connectivity** action.

2. Click the **Jump to code** link.
3. Change the placeholder /**endpoint** to /**region**.
4. Click **Save**.

Add assets

After creating actions and saving your, you can add assets.

An asset represents a physical or virtual device within your organization such as a server, endpoint, router, or firewall. This is where you define the critical specifics for the asset that your app is interacting with. For example, a password for a Google drive or a Microsoft Exchange token.

If you have not already saved your app, you will be required to save your app as a draft app before you can add assets. Exit the alert and click Save.

From the **Console Output** and **Asset** menu bar, complete the following steps:

1. From the **Asset** drop-down menu, select **+ New**.
2. The **Configure New Asset in App Editor** dialog opens. There are five sections, **Asset Info**, **Asset Settings**, **Approval Settings**, **Access Control**, and **Tenants**.
3. Enter the **Asset Info**. Some items, such as the product vendor and product name will populate with information from the App Info entered earlier.
    - Asset name
    - Asset description
    - Product vendor
    - Product name
    - Tags
4. Click the **Asset Settings** tab to modify the asset's settings.
5. Click the right-facing triangle icon to expand the **Advanced** section:
    - Concurrent action limit to cap the number of actions done at the same time.
    - Select any "just in time" credentials the that will be required to use the asset. These are defined in Splunk SOAR (On-premises), and not the App Wizard Editor.
    - Set the user account which will run automated actions.
    - Click **+ Variable** to set any environment variables for holding the specific asset passwords, tokens, or secrets.
6. Click the **Approval Settings** tab to modify the asset's approval requirements.
    1. Set any primary and secondary approvers, if approval of the app's actions is required.
7. Set any **Access Control** settings if your organization requires them to perform actions with this asset.
8. Click the **Tenants** tab to modify the asset's tenant settings, if your organization supports tenants.
9. Set any Tenants settings.
10. Click **Save** to save your changes.

Remove an asset

If you wish to remove an asset created during development or testing follow these steps.

To remove an asset:

1. From the **Home** menu, select **Apps**.
2. Locate your app on one of the tabs, **Configured Apps**, **Unconfigured Apps**, or **Draft Apps**. Apps which have

not yet been published are on the **Draft Apps** tab.
3. Expand the list of configured assets for the app.
4. Locate the asset you want to remove.
5. Click the trash can icon to delete the asset.
6. Click **Delete Asset**.

Example app 'Get IP Info'

The example app needs an asset to connect to ipinfo.io.

1. In the left navigation, click the right-facing triangle icon to expand the **Actions** section.
2. In the the **Console Output** and **Asset** menu bar, select the asset **ipinfo** from the **Asset** drop-down menu.

Test app actions

After creating app actions, you can test the app actions.

1. In the left navigation, click the right-facing triangle icon to expand the **Actions** section.
2. In the the **Console Output** and **Asset** menu bar, select the asset to use for your tests from the **Asset** drop-down menu.
3. Select the action you wish to test, then click the right-facing triangle icon to expand the action's definition.
4. Type any inputs the action will need to run.
5. Click **Test Action**.
6. See the results in the **Console Output** menu.

See Action result.

Jump to code

After creating app actions, you can jump to the code path where the action is defined.

From the action module, click **Jump to code**. This takes you directly to the component file for the action, `_connector.py`.

In order for jump to code to work, the function name must follow the convention `handle<action_name>`. Some apps, such as Maxmind, don't follow this convention, so you can't use jump to code in those scenarios.

If the component file contains errors, you also can't use jump to code in those scenarios.

See App authoring API.

Validate app JSON

A JSON file gets created as you enter your app information, dependencies, configuration details, and actions. Validating the app JSON is optional. You can view it and change it manually if you want to.

To edit the json file, complete the following steps:

1. From App Details, click **Edit JSON**.
2. Make changes.
3. Click the checkmark to validate the JSON syntax and save the file.

App components

A Splunk SOAR app consists of a number of components. The App Editor displays the component files in the UI. See Splunk SOAR (On-premises) app components.

Edit your connector module

Splunk SOAR apps provide connectivity between Splunk SOAR (On-premises) and third-party security products and devices. Connector modules are written in Python and imported into Splunk SOAR as Python modules when packaged within an app.

As you use the App Editor to create configurations, app actions, assets, and so on, these get added to the `_connector.py` file as stubs. This file acts as a template, so you need to edit it. Editing this file is not optional.

See Connector module development.

Define your constants

Define your constants in the `_consts.py` file if necessary. For use when you have hardcoded values to be seen by any function. If you define the constants here, then you do not have to find and replace the values throughout the code if the values change.

Provide detailed information about the app

Update the `readme.html` to provide detailed information for users to read about the app.

Initialize and define a Python package

The `_init_.py` file is required to initialize and define a Python package. You can use an empty file.

Exclude files from the compiler

The `exclude_files.txt` is part of the built-in build process. You do not need to edit this file.

Publish your app

When you have completed development of your app, and are ready to use it, you need to publish it. See Publish an app draft.

Clone an existing app

You can also create a new app by cloning an existing app. Cloning allows you to rename the app, copy the assets, and save the app as a draft. Cloning an existing app creates a new, unique copy of the existing app and does not modify the app which was cloned.

Apps that are not-created through the App Editor can't be edited, but can only be viewed or cloned in the editor. This applies to apps downloaded from Splunkbase. Cloned apps are given their own unique app identifier.

To clone an app, complete the following steps:

1. Navigate to the **Main Menu**.
2. Select **Apps**.
3. From the **Configured Apps** tab, locate the app you want to clone.
4. Click the clone button that looks like two squares.
5. (Conditional) If the app has assets that you want to copy, select which assets to copy.
6. Enter a new name for the App.
7. Select **Copy**.

After cloning, a new copy of the app is saved in the **Draft Apps** tab. The following items are saved:

- Version number
- Actions
- Dependencies
- Configuration details
- Assets
- All app files

Publish an app draft

To publish a draft app, complete the following steps:

1. Navigate to the **Main Menu**.
2. Select **Apps**.
3. From the **Draft Apps** tab, locate the app you want to publish.
4. Click **Publish**.

After publishing, the console output displays, "Install App Successful." The app is saved in the **Configured Apps** tab.

Once an app is published, the following conditions apply.

- Published apps can be viewed in the App Editor, but they cannot be edited. To edit a published app that contains Python code, see Clone an existing app.
- Cloning an app does not modify the existing app. It creates a new, unique draft version of the app.
- New draft versions of a published app have an incremented version number, based on the original app.

Export or import an app

To move an app from one instance to another, you'll export it from one instance and import it to another instance.

To export an app, complete the following steps on an instance of Splunk SOAR (On-premises):

1. Navigate to the **Main Menu**.
2. Select **Apps**.
3. From either the **Draft Apps** tab or **Draft Apps** tab, locate the app you want to export.
4. Edit the app.
5. Click '*Download*.

To import an app, complete the following steps on another instance of Splunk SOAR (On-premises):

1. Navigate to the **Main Menu**.
2. Select **Apps**.
3. Click **Install Apps**.

When you important an app, it is given its own unique identifier app id. Importing an app multiple time will create multiple draft versions of the app, each with their own unique identifier app id.

App Structure

Connector module development

Splunk SOAR apps provide connectivity between Splunk SOAR (On-premises) and third-party security products and devices. Connector modules are written in Python and imported into Splunk SOAR as Python modules when packaged within an app.

When an action is run, the Splunk SOAR action daemon, actiond, invokes an executable that imports the appropriate module and runs the action. When invoked, an IPC connection is opened between the app and actiond. This communication is handled automatically.

App connectors leverage a Splunk SOAR supplied class called BaseConnector that is the main point of contact between the app connector class and Splunk SOAR (On-premises).

Apps need to generate results of an action run. This result is abstracted in another class called ActionResult that the app connector uses.

The following code shows some Splunk SOAR classes:

```
# Phantom imports
import phantom.app as phantom
from phantom.app import BaseConnector
from phantom.app import ActionResult
```

Action workflow

An action is created in Splunk SOAR (On-premises) either through the UI or through a playbook. Depending upon the action and the asset it is run on, one or more apps are chosen and an action JSON containing the asset configuration and action parameters is created for each app that is run. The app connector Python module is loaded up and its member function '_handle_action' is called with the Action JSON as the parameter. A single instance of execution of the app connector is called an app run. Also a single app run occurs for a single action for example, 'block ip' or 'whois domain'.

> Don't use the app connector class with implement _handle_action, use implement handle_action. Doing so allows the BaseConnector to use the action JSON first and it validates the parameters and event before the AppConnector::handle_action is called.

The following code is a snippet of the app JSON containing the metadata describing the configuration and action parameters, for an action called 'whois domain' of a sample whois app.

```
{
    "appid"       : "776ab991-313e-48e7-bccd-e8c9650c239a",
    "name"      : "WHOIS Sample App",
    "description" : "Query WHOIS",
    "publisher": "Phantom",
    "type": "information service",
    "main_module"    : "samplewhois_connector.py",
    "app_version": "1.0",
    "product_vendor": "Generic",
    "product_name": "Whois Server",
```

```
    "product_version_regex": ".*",
    "configuration": {
          "whois_server": {
            "description": "Whois server to use to query",
            "data_type": "string",
            "required": false
          }
    },
    "actions": [
      {
        "action" : "samplewhois domain",
        "description": "Run whois lookup on the given domain",
        "type": "investigate",
        "identifier": "whois_domain",
        "read_only": true,
        "parameters": {
          "domain": {
            "description": "Domain to query",
            "data_type": "string",
            "contains": ["domain name"],
            "primary": true,
            "required": true
          }
        },
        ..
        ..
      }
    ]
}
```

The following is an example of the action JSON containing the configuration and parameter dictionary, for an action called 'whois domain' that is passed from Splunk SOAR (On-premises) to the app.

```
{
   "config" : {
      ...
      ...
      "whois_server" : "whois.arin.net"
   },
   "container_id" : "36f4639d-4574-441b-ac39-8a039a5a8534",
   ...
   ...
   "identifier" : "whois_domain",
   "parameters" : [
      {
         "domain" : "amazon.com"
      },
      {
         "domain" : "google.com"
      }
   ]
}
```

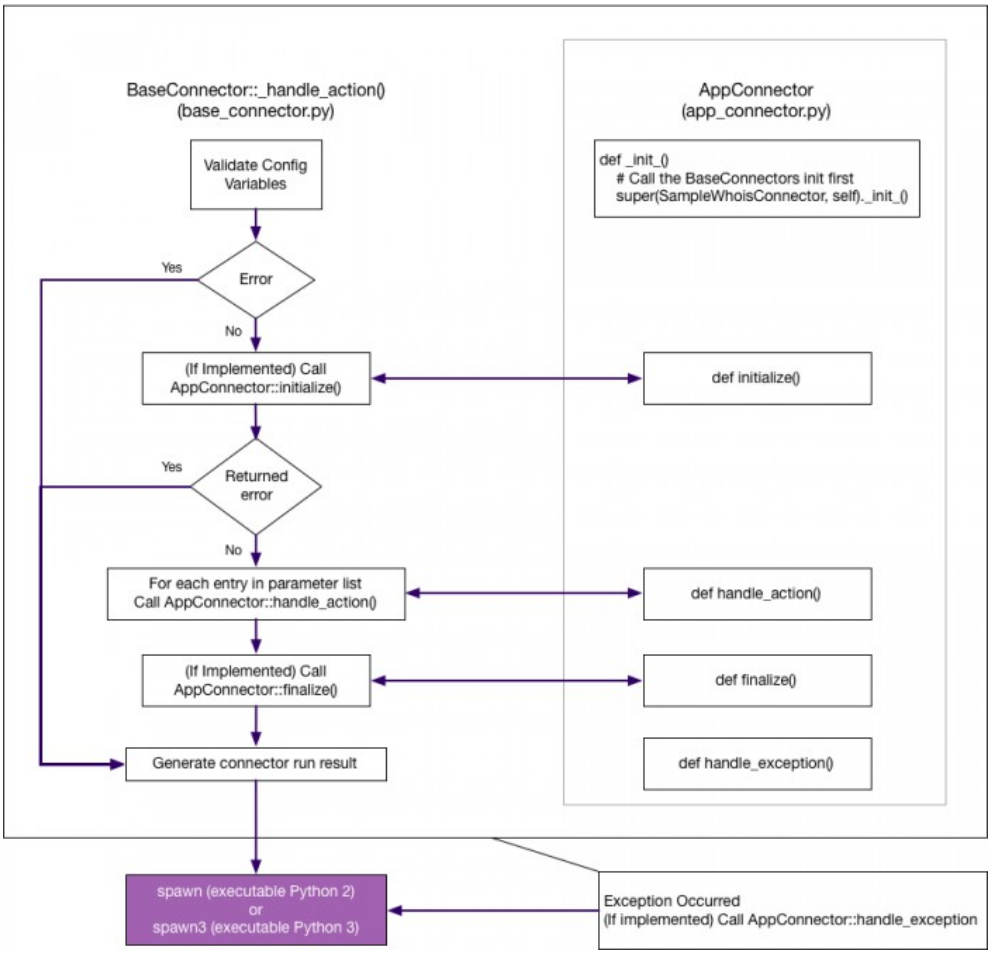
At runtime, configuration and parameter dictionaries are never accessed directly from the app. Call the BaseConnector member functions to get access to these dictionaries.

The parameters key is a list of dictionaries. This is to facilitate multiple actions into one connector run. This means the BaseConnector::_handle_action calls AppConnector::handle_action multiple times, once each with the current parameter dictionary.

```
{
   "domain" : "amazon.com"
}
```
Called a second time with the parameter.

```
{
   "domain" : "google.com"
}
```
BaseConnector::_handle_action

The following diagram represents the calls that occur between the BaseConnector and the AppConnector for a single connector run.



The app API in use by AppConnector is derived from a subclass of BaseConnector. If the app is using Python 2, the Splunk SOAR `Actiond` daemon will route to spawn. If the app is using Python 3, the Splunk SOAR `Actiond` daemon will route to spawn3.

Configuration validation

The BaseConnector::_handle_action parses and loads the app JSON file. This is the file that contains the configuration and parameter variables and the information about which are required. Next, the BaseConnector::_handle_action parses the configuration from 'spawn' and validates it. Before validation it strips all the Python string variable values and deletes the empty values from the configuration dictionary. This way the app writer always gets valid configuration values. In case of an error, the connector result JSON is created and sent back to spawn. If validation succeeds, the AppConnector::initialize() function is called.

AppConnector::initialize()

This function is optional and can be implemented by the AppConnector derived class. Since the configuration dictionary is already validated by the time this function is called, this is a good place to do any extra initialization of internal modules. This function must return a value of either phantom.APP_SUCCESS or phantom.APP_ERROR. If this function returns phantom.APP_ERROR, then the AppConnector::handle_action isn't called. For example, the SMTP app connects to the SMTP server in its initialize function. If this connection fails, then it returns an error and the BaseConnector does not call SmtpConnector::handle_action()

```
def __init__(self):

    # Call the BaseConnectors init first
    super(SmtpConnector, self).__init__()
    self._smtp_conn = None

def initialize(self):

    self._smtp_conn = None

    try:
        status_code = self._connect_to_server()
    except Exception as e:
        return self.set_status(phantom.APP_ERROR, SMTP_ERR_SMTP_CONNECT_TO_SERVER, e)

    return status_code
```

Parameter validation

The App JSON contains information about each action including the parameters that it needs and which are required. BaseConnector uses this information to validate each parameter dictionary element. Similar to the configuration dictionary, string values are stripped and if found empty, deleted. If a required key is not present, this error is recorded for the current parameter dictionary and the call to AppConnector::handle_action is skipped.

Parameter 'contains' validation

Every action can specify the contains of an input parameter in the app JSON. The BaseConnector does extra validation based on this contains value. For example, a parameter that has the contains set as ["ip"], will be validated to be a proper IP address. A parameter can have multiple contains specified for example, ["ip", "domain"]. In this case, the parameter is

considered valid, even if a single validation passes. No extra validation is carried out if the contains is empty or not specified. If the app doesn't accept the validation done by the BaseConnector, there are two options:

1. Add a validator for a particular contains using the set_validator(..) API
2. Implement the BaseConnector::validate_parameters(...) API, which replaces the complete 'contains' based validation of all the parameters done by the BaseConnector.

AppConnector::handle_action()

This function implements the main functionality of the AppConnector. It is called for every parameter dictionary element in the parameters array. In its simplest form, it gets the current action identifier and then calls a member function of its own to handle the action. This function is expected to create the results of the action run that get added to the connector run. The return value of this function isn't used by the BaseConnector. Instead, it loops over the next parameter element in the parameters array and calls the handle_action again.

AppConnector::finalize()

This function gets called once all the parameter dictionary elements are looped over and no more handle_action calls are left to be made. It gives the AppConnector a chance to loop through all the results that were accumulated by multiple handle_action function calls and create any summary, if required. Another usage is cleanup, disconnecting from remote devices and so on.

AppConnector::handle_exception()

All code within BaseConnector::_handle_action is within a 'try: except:' clause. This makes it so if an exception occurs during the execution of this code it is caught at a single place. The resulting exception object is passed to the AppConnector::handle_exception() to do any cleanup of its own, if required. This exception is then added to the connector run result and passed back to spawn, which gets displayed in the Splunk SOAR UI.

All apps should handle exceptions in their code and return back a proper error message.

Result

As noted previously, each connector run can have multiple internal actions that are carried out. In fact, it can have one action for each element of the parameter list. Each individual action can be represented by an instance of the ActionResult class. In the 'whois domain' sample action, two ActionResult objects are created, one for the whois action carried out for 'amazon.com' and another for 'google.com'. Each of these ActionResults are added to the connector result when the action is carried out. Once all the possible calls to AppConnector::handle_action are complete, the BaseConnector creates the connector run result, in JSON format, based off all the ActionResult objects that were added and sends it back to spawn.

One thing to note here is that the app writer does not create the result JSON. As actions are performed, ActionResult objects are created, and if they need to be part of the connector run they are added to the connector run using the BaseConnector interface. The ActionResult class contains member functions the app writer uses to set the status, message, and other information about the action. The creation of the result JSON is done automatically by the BaseConnector.

Connector result

A Connector Run result JSON contains the following parts:

- List of action results. Each entry in this list is represented by an ActionResult object that is added by the app author as and when actions are carried out using BaseConnector::add_action_result(...)
- Status of the connector run, success or failure. This value can be set using the BaseConnector::set_status(...) function. In most cases the app doesn't need to set this value explicitly. The BaseConnector loops through the ActionResult list and if all ActionResult objects have their status set to **failed**, then the connector run is marked as failed. Even if one ActionResult in the list is successful, then the result of the connector run is marked as success.
- Message explaining the connector run result. This value doesn't need to be set by the app. The BaseConnector creates a message containing the number of successful ActionResult objects that are found in the list.
- Summary dictionary. This dictionary is updated by the BaseConnector to contain the total number of actions that ran and how many were successful.

From an app author's point of view, the only part of a connector run that needs to be completed is the ActionResult list.

Action result

Each action result is represented by an object of the ActionResult class. The object encapsulates the following details about an action performed.

Object	Description
Parameter	Every action needs to have a dictionary that represents the parameters that it acted upon and their values. This is required because the input action JSON can have different values than the ones that get operated on by the connector or the connector can add the values of parameters that were not specified. Creating this dictionary allows you to see what values the app used to perform the action. According to the 'whois domain' action the resultant parameter dictionary can be the following: `"parameter" : {` `  "domain" : "amazon.com"` `},` Use `ActionResult::update_param(<dict>)` to update the parameter dictionary. Other functions that operate on the parameter dictionary are `ActionResult::set_param(<dict>)` and `ActionResult::get_param(<dict>)`.
Status	Use `ActionResult::set_status()` to set the status of an ActionResult object. This function also takes an optional message that can be used to set the message of an ActionResult object.
Message	Use `ActionResult::set_status()` or `ActionResult::append_to_message()` functions to update the message of an ActionResult. If the app author does not set the message in an ActionResult object, then the object will create a textual representation of the ActionResult summary dictionary.
Summary	This dictionary contains a summary about the action performed. For the 'whois domain' example this dictionary contains the most interested data about the domain. `"summary" : {` `  "city" : "Reno",` `  "country" : "US",` `  "organization" : "Amazon Technologies, Inc."` `}`
Data	Data is where the whole output of the action performed is added. This always is a list, even when it will contain only a single item, it is still a list. Use ActionResult::add_data to add the action output to the ActionResult. For the 'whois domain' example the 'data' looks like the following sample: `"data" : [` `  {`

Object	Description
	"contacts" : { "admin" : { "city" : "Reno", "country" : "US", "organization" : "Amazon Technologies, Inc.", ... }, "billing" : null, "registrant" : { "city" : "Reno", "country" : "US", ... }, ... "nameservers" : ["ns2.p31.dynect.net", "pdns1.ultradns.net" ....], "raw" : ["Domain Name: amazon.com\nRegistry Domain ID: 281209_DOMAIN_COM-VRSN\nRegistra ....], "registrar" : ["MarkMonitor, Inc."], "status" : ["clientUpdateProhibited (https://www.icann.org/epp#clientUpdateProhibited)", ...], "updated_date" : ["2014-04-30T12:11:08"], "whois_server" : ["whois.markmonitor.com"] } ....]

Output metadata

The output of the action can generate information which is added to the data section of the ActionResult object through the `ActionResult::add_data(...)` function. The 'output' key in the app JSON file describes the content of this data section. Add this key for every action.

Splunk SOAR (On-premises) parses this section and auto generates the documentation so a playbook writer gets all the required information about the output generated by the action. Rendering this data as a table like widget makes for a better view. This can be done easily by filling up certain keys in the 'output' section. Splunk SOAR (On-premises) matches the output of one action to the input parameter of another by matching the 'contains' keyword. This section allows the app author to specify the 'contains' of the data

See Metadata for more information.

App installation

All the required files of an app including the .json and .pyc files need to be placed in a TAR file. The TAR file contains a single directory containing all app files. For example, to create the installer TAR file of an app called samplewhois use the following command:

```
tar -zcvf samplewhois.tgz samplewhois
```

This TAR file can be used to install the app in a Splunk SOAR (On-premises) instance using the install app button found on the apps page.

Installed apps are placed in a sub folder created in the apps directory based on the app name and GUID following this format: `/opt/phantom/apps/appname_<GUID>/`. For example, the samplewhois app is installed in `/opt/phantom/apps/whoissample_<GUID>/`.

PYTHONPATH

As mentioned in previous topics, the connector modules are called into by the spawn executable for every action carried out. Before spawn calls into various connector functions, it sets up the PYTHONPATH, which includes the Splunk SOAR library directory and the app installed directory. For the samplewhois example the path `/opt/phantom/apps/whoissample_<GUID>/` folder will be part of the PYTHONPATH. This allows the app author to distribute modules that are internally used in the app with the app TAR file itself. As long as the app directory TAR filecontains an '__init__.py' (or __init__.pyc) file distributed with it, the app can safely import modules it has distributed with the app. For example, you can place all the utility/helper functions within a set of files and distribute them with the app. The main connector file can then safely import them at runtime.

CA bundles

It's common for apps to use a REST API to communicate with an external device and use the Python requests module to do so. The requests module picks up the CA bundle file pointed to by the REQUESTS_CA_BUNDLE environment variable. Splunk SOAR (On-premises) comes with a CA bundle preinstalled on it through the phantom_cacerts rpm. The bundle file is located in your Splunk SOAR (On-premises) instance at `/opt/phantom/etc/cacerts.pem`. At runtime for every action that is executed, Splunk SOAR (On-premises) will set the the REQUESTS_CA_BUNDLE variable to the `/opt/phantom/etc/cacerts.pem` file before executing the action, so if the app is using the requests module, the CA Bundle will already be set for it to use.

There is a knowledge base article on the Splunk SOAR portal with instructions on how to install and troubleshoot using your own certificates and CAs, which can be found at https://my.phantom.us/kb/16.

Test connectivity

Splunk SOAR (On-premises) allows the user to configure an asset based on the asset configuration defined by an app in the configuration section of the app metadata. The same UI also allows you to test the configuration. This allows the user to see if the configuration is correct or not. This is implemented by the Splunk SOAR UI through a Test Connectivity button on the asset configuration page and by mapping this button to a 'test_asset_connectivity' action in the app. If the app does not implement this action, the TEST CONNECTIVITY button is not displayed. An app can implement this action by adding the following action into its app JSON.

```
{
  "action": "test connectivity",
  "description": "Validate the asset configuration for connectivity",
  "verbose": "This action logs into the device using a REST Api call to check validate the asset
configuration",
  "type": "test",
  "identifier": "test_asset_connectivity",
  "read_only": true,
  "parameters": {
  },
  "output": [],
  "versions":"EQ(*)"
```

```
}
```

Modify the verbose value to specify how the test connectivity is carried out. This action isn't going to be passed any parameters. It needs to work on the asset configuration only. The identifier key used in the previous example is test_asset_connectivity and must be implemented in the connector script. The action type is set to test. This needs to be the case since the UI checks for this value. Also the progress and results of the action are displayed in a dialog box synchronously so it is helpful to be a bit more descriptive by using the self.save_progress() or self.send_progress() function calls.

Ingestion

In Splunk SOAR (On-premises), data sources are services or devices that supply information that you might want to store or act on. An app can support extracting such data from a device and ingesting it into Splunk SOAR by implementing the `on poll` action. An app's ingestion action handler can be called in two instances:

1. The Poll Now button of an asset configuration page.
2. Scheduled ingestion.

In each of the previously mentioned instances the parameters that are passed to the action and the action name identifier that is used are described in the following JSON.

If an app does not add this action to the app JSON, ingestion configuration and interactions like the Poll Now button aren't displayed on the corresponding asset.

```
{
  "action": "on poll",
  "description": "Callback action for the on_poll ingest functionality.",
  "type": "ingest",
  "identifier": "on_poll",
  "read_only": true,
  "parameters": {
    "container_id": {
      "data_type": "string",
      "order": 0,
      "description": "Container IDs to limit the ingestion to.",
      "allow_list": true
    },
    "start_time": {
      "data_type": "numeric",
      "order": 1,
      "description": "Start of time range, in epoch time (milliseconds)",
      "verbose": "If not specified, the default is past 10 days"
    },
    "end_time": {
      "data_type": "numeric",
      "order": 2,
      "description": "End of time range, in epoch time (milliseconds)",
      "verbose": "If not specified, the default is now"
    },
    "container_count": {
      "data_type": "numeric",
      "order": 3,
      "description": "Maximum number of container records to query for."
    },
    "artifact_count": {
      "data_type": "numeric",
      "order": 4,
```

```
      "description": "Maximum number of artifact records to query for."
    }
  },
  "output": [],
  "versions":"EQ(*)"
}
```

Configure metadata in a JSON schema to define your app's configuration

Use a JSON schema to define the configuration of your Splunk SOAR app. The JSON may vary in complexity depending on the sophistication of the app and whether you want to customize how data produced by your app is rendered.

Top level definition

The following top level keys define the primary facets of the app:

```
{
  "appid" : "fd4dc1ce-b63c-4d0e-94e6-df0ae680d99c",
  "name" : "SampleApp",
  "description" : "Query the Sample reputation service",
  "type" : "reputation",
  "main_module" : "sampleapp_connector.py",
  "app_version" : "1.0.0",
  "product_vendor" : "Contoso",
  "product_name" : "Contoso Reputation",
  "product_version_regex": ".*",
  "logo": "contoso.svg",
  "logo_dark": "contoso_dark.svg",
  "min_phantom_version": "1.0.240",
  "publisher" : "Contoso Corp",
  "package_name": "phantom_whois",
  "license": "Copyright (c) 2019 Splunk, Inc.",
  "configuration": {},
  "actions": []
}
```

Key	Required?	Description
appid	Required	The ID of the app for unique identification. This ID is in GUID/UUID version 4 format. It can be generated applications for the required platform such as uuidgen for linux, guidgen.exe for windows, or www.guidgenerator.com on the web.
name	Required	This is the name of the app.
description	Required	This is a more verbose description of the app. This is used to display a description of the app in the user interface, and also for automatically generated documentation on an app.
type	Required	This is the app type. The type is used to group apps at a level higher than the product_vendor name. Some of the types currently defined by Splunk SOAR apps include siem, email, endpoint, firewall, forensic, information, investigative, network access control, reputation, sandbox, threat intel, ticketing, virtualization and generic.
main_module	Required	This is the file name containing the module with the entry point code. It usually contains the action implementations for this app. This file is loaded when an action that this app supports is run. The Python code is located in the current app's directory.
app_version	Required	This is the app version. There may be multiple versions of an app on a given system in order to support different versions of a vendor/product combination. Versions start at 1.0.
pip_dependencies	Optional	Use this to specify pip dependencies that need to be installed for the app to function. See Specifying pip dependencies for more information.

Key	Required?	Description
pip3_dependencies	Optional	Use this key to specify any dependencies that must be installed for your Python 3 app. If this key is present, pip_dependencies is ignored. If this key is not present, but pip_dependencies is, that information will be used for both Python 2 and Python 3.
min_phantom_version	Optional	Use this key to specify the minimum Splunk SOAR (On-premises) version that the app supports. The platform will validate the version during app installation and fail if the minimum criteria is not met.
product_vendor	Required	This is the vendor associated with the product that this app supports. For example, ReversingLabs, Cisco, IBM, FireEye.
product_name	Required	This is the name of the product that this app supports.
product_version_regex	Required	This is the supported version of the product that your app uses. It is a regex and is rendered by the platform in the app documentation. The app requires code to extract and validate the version from the third party device or service that it uses. A '.*' represents all versions.
publisher	Required	This is your name, as the publisher and creator of this app. This is your company name if you are a vendor, or your own name if you are an individual.
package_name	Required	This is the package name that will be used to install the app through the yum/rpm commands.
license	Required	This is the license that the app is released under. This is the license string that is to be displayed in the yum/rpm info commands output.
consolidate_widgets	Optional	If false, the platform will render every action in its own widget, or if true, a single widget will contain all actions consolidated.
configuration	Required	A set of asset configuration variables, defined in more detail in the following sections. Configuration can be empty if none are needed.
url	Optional	This is the URL associated with the app. It is a landing page that gives brief information about the app or extra documentation.
logo	Optional	This is the name of the icon file that is rendered at multiple places in the product in Light Theme. It has to be in the app folder with the rest of the files. SVG, PNG and JPEG files are supported, but SVGs are preferred. The logo must have a transparent background and is represented by the product being integrated. A product icon is preferred over the vendor icon.
logo_dark	Optional	This is the icon that is rendered in Dark Theme.
actions	Required	Detailed specification of the actions that this app supports. This defines the input parameters that the app needs for each action and the output that they generate.
rest_handler	Optional	This is the path to an optional REST handler function.
python_version	Required.	Defines the Python version used to run the app's actions. This value must be "3."

Specifying pip dependencies

You can specify the Python modules that the platform installs during the app installation. See the following example for the format of this dictionary:

```
"pip_dependencies": {
   "pypi": [
      {"module": "requests_ntlm"}
   ],
   "wheel": [
      {"module": "python-rtkit", "input_file": "wheels/python_rtkit-0.7.0-py27-none-any.whl"}
   ]
},
```

The pip_dependencies dictionary supports two keys, but only one of the following keys is required:

1. pypi: A list of modules that need to be installed for the app to work properly. The name to use as a parameter for the pip install is the [module_name] command.
2. wheel: Use wheel to install app dependencies without direct connectivity to the internet. However, all Splunk SOAR apps have this capability. Splunk SOAR (On-premises) runs `pip` with the `--no-deps` parameter during wheel file installation. Each item in the list is a dictionary with the following keys:

- module: The module name.
- input_file: Path to the wheel file relative to the app directory in the app TAR file.

We pull from pypi when the pypi dictionary is defined. See the following for an example of an app install of the requests_ntlm package in both the Python 2 and Python 3 environments:

```
"pip_dependencies": {
   "pypi": [
      {"module": "requests_ntlm"}
   ]
},
"pip3_dependencies": {
  "pypi": [
      {"module": "requests_ntlm"}
   ]
}
```

For Splunk SOAR, Python 3 is required.

Here is an example of an app specifying pip dependencies for Python 2 and Python 3.

```
"pip_dependencies": {
        "wheel": [
            {
                "module": "certifi",
                "input_file": "wheels/py2/certifi-2019.11.28-py2.py3-none-any.whl"
            },
            {
                "module": "chardet",
                "input_file": "wheels/py2/chardet-3.0.4-py2.py3-none-any.whl"
            },
            {
                "module": "geoip2",
                "input_file": "wheels/py2/geoip2-2.9.0-py2.py3-none-any.whl"
            },
            {
                "module": "idna",
                "input_file": "wheels/py2/idna-2.8-py2.py3-none-any.whl"
            },
            {
                "module": "ipaddress",
                "input_file": "wheels/py2/ipaddress-1.0.23-py2.py3-none-any.whl"
            },
            {
                "module": "maxminddb",
                "input_file": "wheels/py2/maxminddb-1.5.1-cp27-none-any.whl"
            },
            {
                "module": "requests",
                "input_file": "wheels/py2/requests-2.22.0-py2.py3-none-any.whl"
            },
```

```
            {
                "module": "urllib3",
                "input_file": "wheels/py2/urllib3-1.25.7-py2.py3-none-any.whl"
            }
        ]
    },
    "pip3_dependencies": {
        "wheel": [
            {
                "module": "certifi",
                "input_file": "wheels/py3/certifi-2019.11.28-py2.py3-none-any.whl"
            },
            {
                "module": "chardet",
                "input_file": "wheels/py3/chardet-3.0.4-py2.py3-none-any.whl"
            },
            {
                "module": "geoip2",
                "input_file": "wheels/py3/geoip2-2.9.0-py2.py3-none-any.whl"
            },
            {
                "module": "idna",
                "input_file": "wheels/py3/idna-2.8-py2.py3-none-any.whl"
            },
            {
                "module": "maxminddb",
                "input_file": "wheels/py3/maxminddb-1.5.1-py3-none-any.whl"
            },
            {
                "module": "requests",
                "input_file": "wheels/py3/requests-2.22.0-py2.py3-none-any.whl"
            },
            {
                "module": "urllib3",
                "input_file": "wheels/py3/urllib3-1.25.7-py2.py3-none-any.whl"
            }
        ]
    },
```

For Splunk SOAR, Python 3 is required.

Configuration Section

In order to run an action, an app must operate on an asset that has been configured by the end user within Splunk SOAR (On-premises). First, the platform must have one or more instances of an asset configured; at least one that is directly supported by the app, and that matches the vendor and product that the app supports.

The configuration section contains variables that are used when configuring an asset, essentially serving as a template for that. As a result, the configuration of an asset is data-driven based on the variables defined here. Each asset vendor/product combination will typically have a single app that operates on it. For example, a single app that supports and operates on Cisco ASA devices, and the configuration variables required by the app to connect to that device are defined here. These are static configuration variables for an asset that serve as a template when configuring the asset under the Administration menu, such as a username, password, or API key as shown in the following example. Dynamic parameters that change each time an app runs an action are defined in the parameters section of the corresponding action and are not configuration variables.

```
{
  ...
  "configuration": {
    "apikey" : {
      "data_type": "string",
      "description": "Contoso API key",
      "required": true
    }
  },
```

The configuration section contains a list of variables (keys) that each contain a dictionary to describe that key. This defines how that variable is rendered and used within the system:

Key	Required?	Description
data_type	Required	The type of variable. Supported types are string, password, numeric, and boolean.
description	Required	A description of this variable. This value is used as a label that is displayed on the UI next to the input control. Keep this brief, preferably a single line, since it is displayed on the UI. If you need to explain the app, or asset configuration in more detail use a readme.html file.
required	Optional	Whether or not the variable is mandatory. If mandatory, the variable must be configured by the user when defining an asset that this variable operates on. Default is False.
value_list	Optional	To allow the user to choose from a pre-defined list of values displayed in a drop-down, specify them as a list. For example, ["one", "two", "three"]
default	Optional	To set the default value of a variable in the UI, use this key. The user will be able to modify this value, so the app will need to validate it. This key also works in conjunction with value_list.
order	Optional	The order key, starting at 0, allows the app author to control the display order of the controls in the UI.

Not all apps require a configuration section. For example, the MaxMind app has no need for one, since there is nothing to configure. It can be left empty as shown by `"configuration": {},`.

During app updates, configuration requirements may change. If the new app version adds a new parameter that is required, the currently configured asset will become invalid and user intervention is required for the app to function. For a smoother upgrade process, the app author tries to set the new asset configuration values as optional and with a default value instead of adding a required value. The default value is specified in the app JSON and is also handled in the app code.

README file

An app author can bundle a readme.html file in the app directory, which the platform renders as part of the app documentation. It is rendered between the app description and the asset configuration parameters.

Place Holder Data Type

The order key allows an app author to specify the order in which the controls are displayed to the user. In order to insert a blank, no control, at a specific location use the ph data_type with the order key. For example, to display a blank space between the first and second control define the configuration as shown in the following example:

```
{
  ...
    "configuration": {
      "server": {
```

```
      "data_type": "string",
      "description": "Server IP/Hostname",
      "order": 0,
      "required": true
    },
    "ph": {
      "data_type": "ph",
      "order": 1
    },
    "username": {
      "data_type": "string",
      "description": "Username",
      "order": 2,
      "required": false
    },
    "password": {
      "data_type": "password",
      "description": "Password",
      "order": 3,
      "required": false
    },
```

This data type is also supported for action parameters.

Actions Section

The actions key defines an array of actions that this app supports. This exposes the core functionality that the app makes available to Splunk SOAR (On-premises).

```
{
  ...
  actions": [
    {
      "action": "file reputation",
      "description": "Queries Contoso for file info",
      "type": "investigate",
      "read_only": true,
```

Parameter	Required?	Description
action	Required	The name of the action. Action names are high level primitives that are used throughout Splunk SOAR (On-premises). They are used in playbooks when an action is requested and in the UI when an action is run.
identifier	Required	A unique identifier for this action. The action key is mapped to this identifier and that is what is passed down to the app. The app code then uses this unique identifier to determine which action to run. The identifier is maximum five characters.
description	Required	A description of this variable. This value is used as a label that is displayed on the UI next to the input control. Keep the description brief, preferably a single line, since it is displayed on the UI. If you want to explain things in more detail regarding a parameter, or the action in general, use the verbose key.
verbose	Optional	Use this key to fill in any information that you want to document about the action. This text supports HTML formatting.
type	Required	The type of action. This is used to organize actions into logical groups based on their purpose. Type must be one of the following: contain, correct, generic, investigate or test. The only test connectivity action uses the test type. The test type of actions are not displayed in the UI in the Run Action dialog.
read_only	Required	Indicates whether the action is "read-only", or in other words, non-destructive. Executing this action will have no adverse affect on the asset. This typically applies to information gathering actions that do not make any configuration changes to an asset.

Parameter	Required?	Description
undo	Optional	The action that reverts this action.

Naming Actions

Splunk SOAR users are used to a particular naming convention for action names. To aid user's understanding, it helps to reuse action names. The following tips can help you understand how to best name actions:

- Review apps to see if they expose the same action. For example, there may be a "block ip" action exposed by a Cisco ASA app, a Palo Alto Networks app, and a Microsoft Windows app. If available, match your action name to an existing name that has the same purpose. This is useful as your app can benefit from Playbooks that already run this action.
- Keep action names as short as possible while still conveying their intent. App names may contain spaces.
- Utilize the app's documentation or the **Run Action** dialog box to study the action names in the product.

Action Section: Versions

The versions key specifies which versions of the product that this action supports. This key contains a regular expression that is matched against a configured asset to find the app and action within that app that best supports a specific asset.

```
"versions": "EQ(*)"
```

The versions key supports a short number of matching functions which can be one of the following:

Function	Description
`EQ()`	Equal: The version matches a specific version: EQ(*), or EQ(12.*)
`NEQ()`	Not Equal: The version does not match a specific version: NEQ(12.*)
`IN()`	In: The version matches a list of comma separated versions: IN(12.*, 13.*)
`NIN()`	Not In: The version is not in a list of comma separated versions: NIN(12.*, 13.*)

Version matching is used to find the best app to run on a given asset, while allowing you to maintain support for various versions of a vendor's product.

There are two ways to handle multiple versions of a product. You can create a new action within your JSON which has a different version expression and calls a different function within your app. In this case, your app code has two different functions each one supporting a different version or versions of a product. Or, you can create a separate JSON which contains the required metadata for the version that you support. In this case, you can have a separate app altogether, with its own appid; one that supports only the versions specified in this separate JSON.

Action Section: Parameters

Within an action entry, the parameters key defines a list of parameters that are passed to this action. These parameters are typically specified in automation playbooks and pass through to the app when an action is run. The name of the key represents the parameter name. For example, in the following example hash is the name of the parameter. Each action and its parameters are documented on the platform, take a look at the names used by action parameters of the same action in other apps. For example, an action like list processes take as input an ip address or host name named ip_hostname, another example domain reputation takes a parameter named domain.

```
"parameters": {
   "hash": {
```

```
      "description": "the hash of the file to be queried",
      "data_type": "string",
      "contains": ["hash", "sha256", "sha1", "md5"],
      "required": true,
      "allow_list": true
   }
},
```

Parameter	Required?	Description
description	Required	A short description of this parameter. The description is shown in the user interface when running an action manually.
data_type	Required	The type of variable. Supported types are string, password, numeric, boolean.
contains	Optional	Specifies what kind of content this field contains.
required	Optional	Whether or not this parameter is mandatory for this action to function. If this parameter is not provided, the action fails.
primary	Optional	Specifies if the action acts primarily on this parameter or not. It is used in conjunction with the contains field to display a list of contextual actions where the user clicks on a piece of data in the UI.
value_list	Optional	To allow the user to choose from a pre-defined list of values displayed in a drop-down for this parameter, specify them as a list for example, ["one", "two", "three"]. An action can be run from the playbook, in which case the user can pass an arbitrary value for the parameter, so the app needs to validate this parameter on its own.
default	Optional	To set the default value of a variable in the UI, use this key. The user will be able to modify this value, so the app will need to validate it. This key also works in conjunction with value_list.
order	Optional	The order key, starting at 0, allows the app author to control the display order of the controls in the UI.
allow_list	Optional	Use this key to specify if the parameter supports specifying multiple values as a comma separated string.

Even before the action is passed onto the app, it reaches the BaseConnector first. The BaseConnector parses the app JSON and matches it against the configuration and action parameters that were passed from the Splunk SOAR Core. If the BaseConnector finds that any of the parameters marked as required are missing, it reports an error back to the Splunk SOAR Core. This allows the app writer to write code without checking for the presence of a required parameter.

Action Section: Synchronization

Within an action entry, the optional lock key defines a set of parameters that app authors can set to synchronize actions.

- A lock is represented by its name.
- Multiple actions locking on the same name will be serialized even if the actions are from different apps.
- In the absence of a lock dictionary, the platform will synchronize the action using the asset as the lock name.
- To disable synchronization for an action, the lock dictionary must be present and the 'enabled' key set to false.

```
"lock": {
   "enabled": true,
   "data_path": "parameters.hash",
   "timeout": 600
}
```

Parameter	Required?	Description
enabled	Required	Boolean value that specifies if the lock is enabled or not for this action.
data_path	Optional	The name of the lock. Only valid if lock is enabled. This value can be one of three things. Either a data path that points to a parameter of the action for e.g. "parameters.hash" where 'hash' is one of the parameters of the action,

Parameter	Required?	Description
		or a data path that points to a config parameter for e.g. "configuration.server". At runtime the platform will read the value stored in these data paths and use it as a name of the lock. The third option that the app author can use is a constant string. I.e. any string that does not start with "configuration." or "parameters.". The platform will use this value as is. In case the data_path is not specified, the asset will be used as the lock name.
timeout	Optional	Specifies the number of seconds to wait to acquire the lock, before an error condition is reported.

Action Section: Render key

Within an action entry, the render key specifies how output from this app will be displayed to the user within Splunk SOAR (On-premises). This section is required. Output is typically displayed in two places. First, in the results from an action when viewing a specific action, and second in the Investigation panel of the product. In Investigation, the output is encapsulated within a widget container that can be displayed, expanded, and moved by the user when managing security operations.

The value associated with this key is a dictionary which must contain a key called type.

```
"render": {
   "type": "custom",
   "width": 4,
   "height": 5,
   "view": "sampleapp_view.file_reputation",
   "title": "FILE REPUTATION",
   "menu_name": "SampleApp File Lookup"
}
```

Parameter	Required?	Description
type	Required	This specifies the type of render function to use when displaying output from this app. Splunk SOAR (On-premises) supports a number of built in widgets that display data and also gives you the ability to define your own view to render data from your app in any form you want. Values are: json, table, or custom. The default is table.
width	Required	Specifies the width, in columns, of the container that houses this content. The Investigation screen is 12 columns wide and widgets can be anywhere from one to six columns in width.
height	Required	Specifies the height, in rows, of the container that houses this content. The height must be between 0 and 10.
view	Optional	Specifies the custom view to load when rendering data produced by this app. The format of this value is file.function, referring to the Django Python view file that you authored, and the function for Splunk SOAR to call when rendering your view.
title	Required	Specifies the title that is displayed within the widget header on the Investigation screen.
menu_name	Optional	Specifies the name to be displayed in the widget menu on the Investigation screen when listing this widget.

Action Section: Output

Within an action entry, the output key defines an array of results that are generated by this app. Each member of the array is one variable that matches the corresponding variable in the CommandResult class that the Python app code produces.

```
"output":[
   {
      "data_path": "action_result.data.*.permalink",
      "data_type": "string"
   },
   {
   "data_path": "action_result.data.*.sha1",
```

```
"data_type": "string",
"contains": ["sha1"],
"example_values": [
   "f0eb87677a689a88b58a0613dfd7252fb850393c",
   "d969e7e0b0571370cd6763192bc24ac56c255472"
   ]
},
```

Parameter	Required?	Description
data_path	Optional	Specifies the data path of this field. Data paths are a method of indexing into the JSON in an abstract fashion that allows others who want to access that data to do so by specifying the appropriate path. The data path specified here must be populated by the app code that creates and returns this field.
data_type	Required	The type of variable. Supported types are string, password, numeric, boolean.
contains	optional	Specifies what kind of content this field contains. The purpose of contains is to allow matching of output from one action to another action's parameters.
column_name	Optional	If the output from this app is rendered in the default table widget, when render "type" is table, then column_name specifies the name of this column within the rendered widget when it is displayed. This is a convenient way to show data in Investigation and when viewer action results in tabular form.
column_order	Optional	If the output from this app is to be rendered in the default table widget, when render "type" is table, then column_order specifies the order of this column. Column ordering starts at 0.
example_values	Optional	A list of example values for the data path. These are displayed in the app's documentation page.
map_info	Optional	If the output from this app is to be rendered in the default map widget, when render "type" is map, then map_info specifies the name of this field within the details for this item.

Use the contains parameter to configure contextual actions

Splunk SOAR apps have a parameter for action inputs and outputs called "contains". The contains types, in conjunction with the primary parameter property, are used to enable contextual actions in the Splunk SOAR (On-premises) user interface. A common example is the contains type "ip". This represents an ip address. You might run an action that produces an ip address as one of its output items. Or, you may have ingested an artifact of type ip. If you view the ip in Investigation, you will get a context menu that lets you then run other actions that take ip as an input. When an author is creating an app, they specify that a given data field "contains" an ip, so that Splunk SOAR (On-premises) knows how to treat this piece of data.

Once a data type has been defined as "ip", the platform parses all the actions for all the apps that are installed and it shortlists all the actions that have specified "ip" as one of the contains for a parameter that was marked as primary. These actions will be made available from the context menu for that item.

This is a powerful feature that the platform provides, as it allows the user to chain the output of one action as input to another. As an app author, check that your data type isn't already covered by an existing contains that other apps use before creating a new one for their app. Contains is a list, and a given field may have more than one simultaneous contains type. A common example is a SHA256 which will often be listed both as "sha256" as well as "hash". But, some common concepts can be product specific, such as an "id". While the concept of an ID is generic, in terms of making use of it, an ID from one product generally doesn't work well in a different product.

Besides apps, Playbooks can also add artifacts to their container through the phantom.add_artifact call. Artifacts have a contains type, either by virtue of their CEF type, or by directly specifying a contains type.

The contains types applies to files in the container, such as apk, doc, jar, os memory dump, pdf, pe file, ppt, and xls. Apps and Playbooks can specify a contains on a file. Splunk SOAR (On-premises) will also attempt to determine the file type for manually uploaded files as some Apps, most notable those that implement a detonate file, only handle certain file types.

Since new apps can provide new contains types, this list may differ from what is available on your Splunk SOAR (On-premises) instance. To see the current contains list on a given Splunk SOAR (On-premises) instance, use the REST endpoint /rest/cef_metadata . This displays both the current contains types as well as CEF types and what contains types they map to.

```
anubis task id
apk
carbon black query
carbon black query type
carbon black sensor id
carbon black watchlist
cuckoo task id
cyphort event id
doc
domain
email
file name
file path
file size
firewall rule name
flash
hash
host name
ip
isightpartners report id
jar
javascript
jira project key
jira ticket key
jira ticket status
lastline task id
mac address
malwr task id
md5
mobileiron device uuid
network application
os memory dump
pdf
pe file
pid
port
ppt
process name
qradar offense id
rt queue
rt ticket id
servicenow ticket id
sha1
sha256
srp guid
tanium question
threatgrid task id
url
urlquery queue id
urlquery report id
user name
vault id
vm
volatility profile
wepawet task id
```

```
wildfire task id
xls
```

App authoring API

The Splunk SOAR app authoring API can help you write applications that interact with external devices and create structured results that are then passed onto the Splunk SOAR core to be displayed in the UI and consumed by Playbooks. The complete set of APIs can be logically broken up into the following:

- BaseConnector
- ActionResult
- Vault

BaseConnector

`add_action_result(action_result)`

Add an ActionResult object into the connector run result. Returns the object added.

Parameter	Required	Description
action_result	Required	The ActionResult object to add to the connector run.

`append_to_message(message)`

Appends a string to the current result message.

Parameter	Required	Description
message	Required	The string that is to be appended to the existing message.

`debug_print(tag, dump_object='')`

Dumps a pretty printed version of the 'dump_object' in the <syslog>/phantom/spawn.log file, where <syslog> typically is /var/log/.

Parameter	Required	Description
tag	Required	The string that is prefixed before the dump_object is dumped.
dump_object	Optional	The dump_object to dump. If the object is a list, dictionary and so on it is automatically pretty printed.

`error_print(tag, dump_object='')`

Dumps an ERROR as a pretty printed version of the 'dump_object' in the <syslog>/phantom/spawn.log file, where <syslog> typically is /var/log/. Refrain from using this API to dump an error that is handled by the App. By default the log level of the platform is set to ERROR.

Parameter	Required	Description
tag	Required	The string that is prefixed before the dump_object is dumped.
dump_object	Optional	The dump_object to dump. If the object is a list, dictionary and so on it is automatically pretty printed.

`finalize()`

Optional function that can be implemented by the AppConnector. Called by the BaseConnector once all the elements in the parameter list are processed.

`get_action_identifier()`
: Returns the action identifier that the AppConnector is supposed to run.

`get_action_results()`
: Returns the list of ActionResult objects added to the connector run.

`get_app_config()`
: Returns the app configuration dictionary.

`get_app_id()`
: Returns the appid of the app that was specified in the app JSON.

`get_app_json()`
: Returns the complete app JSON as a dictionary.

`get_asset_id()`
: Returns the current asset ID passed in the connector run action JSON.

`get_ca_bundle()`
: Returns the current CA bundle file.

`get_config()`
: Returns the current connector run configuration dictionary.

`get_connector_id()`
: Returns the appid of the app that was specified in the app JSON.

`get_container_id()`
: Returns the current container ID passed in the connector run action JSON.

`get_container_info(container_id=None)`
: Returns info about the container. If container_id is not passed, returns info about the current container.

`get_current_param()`
: Returns the current parameter dictionary that the app is working on.

`get_product_installation_id()`
: Returns the unique ID of the Splunk SOAR (On-premises) product installation.

`get_product_version()`
: Returns the version of Splunk SOAR (On-premises).

`get_state()`
: Get the current state dictionary of the asset. Will return None if load_state() has not been previously called.

`get_state_file_path()`
: Get the full current state file path.

`get_state_dir()`
: An app might require to create files to access during action executions. It can use the state directory returned by this API to store such files.

`get_status()`
: Get the current status of the connector run. Returns either phantom.APP_SUCCESS or phantom.APP_ERROR.

`get_status_message()`

Get the current status message of the connector run.

`handle_action(param)`

Every AppConnector is required to implement this function It is called for every parameter dictionary in the parameter list.

Parameter	Required	Description
param	Required	The current parameter dictionary that needs to be acted on.

`handle_cancel()`

Optional function that can be implemented by the AppConnector. This is called if the action was cancelled.

`handle_exception(exception)`

Optional function that can be implemented by the AppConnector. Called if the BaseConnector::_handle_action function code throws an exception that is not handled.

Parameter	Required	Description
exception	Required	The Python exception object.

`initialize()`

Optional function that can be implemented by the AppConnector. This is called once before starting the parameter list iteration, for example, before the first call to AppConnector::handle_action()

`is_action_cancelled()`

Returns 'True' if the connector run was cancelled. Otherwise, it returns as 'False'.

`is_fail()`

Returns 'True' if the status of the connector run result is failure. Otherwise, it returns as 'False'.

`is_poll_now()`

The on_poll action is called during **Poll Now** and scheduled polling. Returns 'True' if the current on_poll is run through the **Poll Now** button. Otherwise, it returns as 'False'.

`is_success()`

Returns 'True' if the status of the Connector Run result is success. Otherwise, it returns as 'False'.

`load_state()`

Load the current state file into the state dictionary. If a state file does not exist, it creates one with the app_version field. This returns the state dictionary. If an error occurs, this returns None.

`remove_action_result(action_result)`

Remove an ActionResult object from the connector run result. Returns the removed object.

Parameter	Required	Description
action_result	Required	The ActionResult object that is to be removed from the connector run.

`save_artifact(artifact)`

Save an artifact to Splunk SOAR (On-premises).

Parameter	Required	Description
artifact	Required	Dictionary containing information about an artifact.

Returns the following tuple

Parameter	Description
status	phantom.APP_SUCCESS or phantom.APP_ERROR.
message	Status message.
id	Saved artifact ID if success.

Save artifacts more efficiently with the save_container API, since a single function call can add a container and all its artifacts.
See Active playbooks to learn how to set the run_automation key in the artifact dictionary.

`save_artifacts(artifacts)`

Save a list of artifacts to Splunk SOAR (On-premises).

Parameter	Required	Description
artifacts	Required	A list of dictionaries that each contain artifact data. Don't set the run_automation key for the any artifacts as the API will automatically set this value to 'False' for all but the last artifact in the list to start any active playbooks after the last artifact is ingested.

Returns the following tuple:

Parameter	Description
status	phantom.APP_SUCCESS or phantom.APP_ERROR.
message	Status message.
id_list	List of saved artifact IDs if successful; none otherwise.

Save artifacts more efficiently with the save_container API, since a single function call can add a container and all its artifacts.
See Active playbooks to learn how to set the run_automation key in the artifact dictionary.

`save_container(container)`

Save a container and artifacts to Splunk SOAR (On-premises).

Parameter	Required	Description
container	Required	Dictionary containing info about a container. To ingest a container and artifacts in a single call, add a key called artifacts to the container dictionary. This key contains a list of dictionaries, each item in the list representing a single artifact. Don't set the run_automation key for the container or artifacts. The API will automatically set this value to 'False' for all but the last artifact in the list to start any active playbooks after the last artifact is ingested.

Returns the following tuple

Parameter	Description
status	phantom.APP_SUCCESS or phantom.APP_ERROR.
message	Status message.
id	Saved container ID if success, none otherwise.

See Active playbooks to learn how to set the run_automation key in the container dictionary.

`save_containers(containers)`

Save a list of containers to the phantom platform.

Parameter	Required	Description
containers	Required	A list of dictionaries that each contain information about a container. Each dictionary follows the same rules as the input to save_container.

Returns the following tuple:

Parameter	Description
status	phantom.APP_SUCCESS or phantom.APP_ERROR. This is successful when at least one container is successfully created.
message	Status message
container_responses	A list of responses for each container. There will be one response for each container in the input list. This is a dictionary with the following keys: success (phantom.APP_SUCCESS or phantom.APP_ERROR), message, and id.

This is the most performant API to use during ingestion as you can create all the artifacts and containers at once.

As long as at least one container is successfully added, the status is success. During ingestion, BaseConnector automatically keeps track of how many containers and artifacts are successfully added, though if you do want to know more about an individual failure you need to iterate over the list in the response and check the message.

`save_progress(progress_str_const, *unnamed_format_args, **named_format_args)`

This function sends a progress message to the Splunk SOAR core, which is saved in persistent storage.

Parameter	Required	Description
progress_str_const	Optional	The progress message to send to the Spunk Phantom core. Typically, this is a short description of the current task.
unnamed_format_args	Optional	The various parameters that need to be formatted into the progress_str_config string.
named_format_args	Optional	The various parameters that need to be formatted into the progress_str_config string.

`save_state(state_dict)`

Write a given dictionary to a state file that can be loaded during future app runs. This is especially crucial with ingestion apps. The saved state is unique per asset. An app_version field will be added to the dictionary before saving.

Parameter	Required	Description

state_dict	Required	The dictionary to write to the state file.

`send_progress(progress_str_const, *unnamed_format_args, **named_format_args)`

This function sends a progress message to the Splunk SOAR core. It is written to persistent storage, but is overwritten by the message that comes in through the next send_progress call. Use this function to send messages that need not be stored over a period of time like percent completion messages while downloading a file.

Parameter	Required	Description
progress_str_const	Optional	The progress message to send to the Splunk SOAR core. Typically, this is a short description of the current task being carried out.
unnamed_format_args	Optional	The various parameters that need to be formatted into the progress_str_config string.
named_format_args	Optional	The various parameters that need to be formatted into the progress_str_config string.

`set_status(status_code, status_message='', exception=None, *unnamed_format_args, **named_format_args)`

Sets the status of the connector run result, phantom.APP_SUCCESS or phantom.APP_ERROR. Optionally, you can set the message. If an exception object is specified, it is recorded in the connector run result. It will replace any status and message previously saved in the object. Returns the status_code set.

Parameter	Required	Description
status_code	Required	The status to set of the connector run. It is either phantom.APP_SUCCESS or phantom.APP_ERROR.
status_message	Optional	The message to set. Typically, this is a short description of the error or success.
exception	Optional	The Python exception object that has occurred. BaseConnector will convert this exception object to string format and append to the message.
unnamed_format_args	Optional	The various parameters that need to be formatted into the status_message string.
named_format_args	Optional	The various parameters that need to be formatted into the status_message string.

`set_status_save_progress(status_code, status_message='', exception=None, *unnamed_format_args, **named_format_args)`

Helper function that sets the status of the connector run. This needs to be phantom.APP_SUCCESS or phantom.APP_ERROR. This function sends a persistent progress message to the Splunk SOAR Core in a single call. Returns the status_code set.

Parameter	Required	Description
status_code	Required	The status to set of the connector run. It is either phantom.APP_SUCCESS or phantom.APP_ERROR.
status_message	Optional	The message to set. Typically, this is a short description of the error or success.
exception	Optional	The Python exception object that has occurred. BaseConnector will convert this exception object to string format and append it to the message.
unnamed_format_args	Optional	The various parameters that need to be formatted into the status_message string.
named_format_args	Optional	The various parameters that need to be formatted into the status_message string.

`set_validator(contains, validator)`

Set the validator of a particular contains, this is set for the current connector run only. Call this from the initialize function.
Returns the following tuple:

Parameter	Description
status	phantom.APP_SUCCESS or phantom.APP_ERROR.
message	Status message.

`validate_parameters(param)`
BaseConnector uses this function to validate the current parameter dictionary based on the contains of the parameter. The AppConnector can override it to specify its own validations.

`update_summary(summary)`
Update the connector run summary dictionary with the passed dictionary.

Parameter	Required	Description
summary	Required	The Python dictionary that needs to be updated to the current connector run summary.

Active playbooks

In Splunk SOAR (On-premises), playbooks marked as active are automatically run when a container is added or updated. The addition of an artifact to a container is considered updating a container.

The container and artifact dictionaries support a key named run_automation. When this is set to 'True' active playbooks are run. For best results, set run_automation to 'True' only for the last artifact of the container.

The save_container and save_artifacts APIs set the run_automation key to 'False' for all except the last artifact, so as to run the playbook once per container when all the artifacts are ingested. These APIs are recommended over save_artifact.

The save_artifact API sets the run_automation key to 'False' if not set.

ActionResult

`add_data(item)`
Add a data item as a dictionary to the list. Returns the item added.

Parameter	Required	Description
item	required	This is a dictionary that needs to be added as a new element to the current data list.

`update_data(item)`
Extend the data list with elements in the item list.

Parameter	Required	Description
item	required	This is a list of items that needs to be added to the current data list.

`get_data()`

Get the current data list.

`get_data_size()`
Get the current data list size.

`add_debug_data(item)`
Add a debug data item to the list. The item will be converted to a string object through the str(...) call before it is added to the list. This list is dumped in the spawn.log file if the action result fails.

`get_debug_data()`
Get the current debug data list.

`get_debug_data_size()`
Get the current debug data list size.

`add_extra_data(item)`
Add an extra data item as a dictionary to the list.

Extra data is different from data that is added through the add_data(...) API. Apps can add data as extra data if the data is huge and none of it is rendered. Typically, this is something that needs to be recorded and can potentially be used from a playbook, but not rendered. Returns the item added.

`get_extra_data()`
Get the current extra data list.

`get_extra_data_size()`
Get the current extra data list size.

`update_extra_data(item)`
Extend the extra data list with elements in the item list.

`add_exception_details(exception)`
Add details of an exception into the action result. These details are appended to the message in the resultant dictionary. Returns the current status code of the object.

Parameter	Required	Description
exception	optional	The Python exception object that has occurred. ActionResult will convert this exception object to string format and append it to the message.

`append_to_message(message_str)`
Append the text to the message

Parameter	Required	Description
message	required	The string to append to the existing message.

`get_dict()`
Create a dictionary from the current state of the object. This is usually called from BaseConnector.

`get_message()`
Get the current action result message.

`get_param()`

Get the current parameter dictionary.

`get_status()`
Get the current result. It returns either phantom.APP_SUCCESS or phantom.APP_ERROR.

`get_summary()`
Return the current summary dictionary.

`is_fail()`
Returns 'True' if the ActionResult status is a failure.

`is_success()`
Returns 'True' if the ActionResult status is success.

`set_status(status_code, status_message='', exception=None, *unnamed_format_args, **named_format_args)`
Set the status of a result. To call this function with unnamed format arguments, pass a message. Pass "" as the status_message, if you want to set an exception parameter instead of a message. Pass 'None' if an exception is not to be used. status_code has to be phantom.APP_SUCCESS or phantom.APP_ERROR. Returns the status_code set.

Parameter	Required	Description
status_code	Required	The status of the connector run. It is either phantom.APP_SUCCESS or phantom.APP_ERROR.
status_message	Optional	The message to set. Typically, this is a short description of the error or success.
exception	optional	The Python exception object that has occurred. BaseConnector will convert this exception object to string format and append it to the message.
unnamed_format_args	Optional	The various parameters that need to be formatted into the status_message string.
named_format_args	Optional	The various parameters that need to be formatted into the status_message string.

`set_param(param_dict)`
Set the parameter dictionary with the passed param_dict. This overwrites the current parameter dictionary

Parameter	Required	Description
param_dict	required	The Python dictionary that overwrites the current parameter.

`update_param(param_dict)`
Update the parameter dictionary with 'param_dict'

Parameter	Required	Description
param_dict	Required	The Python dictionary that is to be updated into the current parameter.

`set_summary(summary)`
Replace the summary with the passed summary dictionary. Returns the summary set.

Parameter	Required	Description
summary	Required	The Python dictionary that overwrites the current summary.

`update_summary(summary)`

Updates the summary with the passed summary dictionary. Returns the updated summary.

Parameter	Required	Description
summary	required	The Python dictionary that updates the current summary.

Vault

During the execution of any action other than test connectivity, the app can add attachments to a container. These files are placed in the vault, which is a location within each container. The app requires the container id to add a file in the vault. The BaseConnector::get_container_id() API can be used to get this information.

Every vault item within a container is denoted by a vault_id.

`vault_add(container=None, file_location=None, file_name=None, metadata=None, trace=False):`

Add an ActionResult object into the Connector Run result. Returns the object added.

Parameter	Required	Description
container	Required	The container to add the attachment to. The return value of BaseConnector::get_container_id() is sufficient.
file_location	Required	This is the location of the file on the Splunk SOAR (On-premises) file system. The file has to be written to the path returned by Vault.get_vault_tmp_dir() before calling this API.
file_name	Optional	The file name to use.
metadata	Optional	A dictionary containing metadata information the app can set about this attachment, dictionary keys that can be set are 'size' in bytes, 'contains' is the contains of the file, 'action' is the action name, returned through the BaseConnector::get_action_name() API) and 'app_run_id', which is the unique ID of this particular app run, returned by BaseConnector::get_app_run_id()).
trace	Optional	Set to 'True' to return debug information.

`create_attachment(file_contents, container_id, file_name=None, metadata=None)`

Create a file with the specified contents, and add that to the vault. Returns the object added. Other than accepting file_contents instead of a file_location, this API is the same as vault_add.

Parameter	Required	Description
file_contents	required	The contents of the file. A temporary file with these contents will be created, which will then be added to the vault.
container_id	required	This is the container to add the attachment to. The return value of BaseConnector::get_container_id() is sufficient.
file_name	optional	The file name to use.
metadata	optional	A dictionary containing metadata information the app can set about this attachment, dictionary keys that can be set are 'size' in bytes, 'contains' is the contains of the file, 'action' is the action name, returned through the BaseConnector::get_action_name() API) and 'app_run_id', which is the unique ID of this particular app run, returned by BaseConnector::get_app_run_id()).

`get_vault_tmp_dir()`

Returns the path to the vault's temporary file directory. Files have to be added here before calling vault_add.

`vault_info(vault_id=None, file_name=None, container_id=None, trace=False)`

The vault_info API returns a tuple; of a success flag either True or False, any response messages as strings, and information for all vault items that match either of the input parameters as a list. If neither of the parameters are specified, an empty list is returned.

Parameter	Required?	Description
vault_id	Optional	The alphanumeric file hash of the vault file, such as `41c4e1e9abe08b218f5ea60d8ae41a5f523e7534`.
file_name	Optional	The file name of the vault file.
container_id	Optional	Container ID to query vault items for. To get the current container_id value from an app that is executing an action, use the return value of BaseConnector::get_container_id().
trace	Optional	Set to 'True' to return debug information.

Usage examples

Import the vault APIs:

```
from phantom import vault
```

Add a file to the vault:

```
# Container should be a container dict or a container id

success, message, vault_id = vault.vault_add(container=container,
file_location='/opt/phantom/somewhere/over/the/rainbow', file_name='Low Latency Expert.docx',
metadata=None, trace=False)

assert(success)
```

Get information about a file in the vault:

```
success, message, info = vault.vault_info(vault_id=vault_id, container_id=container['id'], trace=True):
assert(success)
print(info)

info contains:
[{'id': 8, 'created_via': 'automation', 'container': 'msg', 'task': '', 'create_time': '39 minutes ago',
'name': 'Low Latency Expert.docx', 'user': 'admin', 'vault_document': 8, 'mime_type': 'application/msword',
'hash': '0c96c0561edabf40928e49f0589b9bf41ef70fea', 'vault_id': '0c96c0561edabf40928e49f0589b9bf41ef70fea',
'size': 79822, 'path': '/opt/phantom/vault/0c/96/0c96c0561edabf40928e49f0589b9bf41ef70fea', 'metadata':
{'md5': '947e1761e31d16061d696c5d79be465f', 'sha1': '0c96c0561edabf40928e49f0589b9bf41ef70fea', 'sha256':
'6e14992c829110e4bb672b7b496620be0b1f6b868e5b84e27e9ad28c33f10aaa'}, 'aka': ['Low Latency Expert.docx'],
'container_id': 5, 'contains': ['doc', 'vault id']}]
```

Deprecated functions

The following functions are deprecated. Use the equivalent vault class member functions.

Deprecated API	Supported API
add_attachment	vault_add
get_vault_file	vault_info
get_meta_by_hash	vault_info

Deprecated API	Supported API
get_file_info	vault_info
get_file_path	vault_info

Deprecated API definitions

`get_file_path(vault_id)`

Returns the full path of the vault file on the filesystem, given the vault_id. This function replaces the get_vault_file(...) function.

This function has been deprecated. Use vault_info instead.

Parameter	Required	Description
vault_id	required	The vault_id of the vault file.

`get_file_info(vault_id=None, file_name=None, container_id=None)`

Returns information on all vault items that match either of the input parameters. This function replaced the get_meta_by_hash(...) function. If neither of the parameters are specified, the return value will be an empty list.

This function has been deprecated. Use vault_info instead.

Parameter	Required	Description
vault_id	required	The vault_id of the vault file.
file_name	optional	The name of the vault file.
container_id	optional	Container ID to query vault items for. To get the current container_id value from an app which is executing an action, use the return value of BaseConnector::get_container_id().

Use data paths to present data to the Splunk SOAR (On-premises) web interface

Data paths are a way for each command to pass down information about the data that needs to be presented to the UI layer. Data paths have the following supported starting locations:

Location	Description
`action_result.data`	Used to access the data added to a CommandResult.
`action_result.message`	Used to access the message added to a CommandResult.
`action_result.parameter`	Used to access the parameters passed to a command.
`action_result.status`	Used to access the status from a CommandResult.
`action_result.summary`	Used to access the summary information added to the CommandResult.
`artifact`	Used to access artifacts in a container when authoring playbooks.
`summary`	Used to access the summary of the entire connector operation.

Each starting point is a collection. The `action_result.data` collection is always an array. Each element of this array is an object of arbitrary complexity added by the connector. Whoever creates the data path must understand the data structure

used by the connector. Both the `action_result.summary` and `action_result.summary` starting points reference a dictionary. While a connector writer can potentially add complex data structures to the summary, the intended purpose of these areas are for simple summary data and the view writer will typically not have to go more than one level into the summary areas.

The path separator is the . character. The path may be constructed out of the following data types:

Data type	Description
Non-numeric text characters	If the path segment is a non-numeric value, then the container must be a dictionary and the path segment is assumed to be a key in the dictionary.
Numeric characters	If the path is strictly numeric, then the container is expected to be an array and the path segment accesses the 0-based index into the array. While it is possible that a dictionary with a numeric key can work, it is discouraged and is not guaranteed.
An empty string	An empty path segment accesses all items in the container. Either all elements in the array or all values in a dictionary. Typically this is used to construct a view over multiple entries in the action_result.data. Since the action_result.data is an array and widgets expect to show multiple results, this allows the widget to access all the individual results.

Consider the data path `action_result.data.*.information.0.leaf`. The following data structure is added to each CommandResult:

```
{
  "information": [
    {
      "leaf": "important information is here!",
      "other_data": ...
    },
    {
       "other data": ...
    }
  ]
  "other data": ...
}
```

This path is accessing the `action_result.data`. For every command result it is accessing the information key. The information key is an array and the path is explicitly only accessing the 0th element of the array. Inside the array are dictionaries in which you find a key called leaf which contains the data the widget renders.

Specifying an invalid path attempting to index into an array will cause an exception and the widget will not render.

Only data for fully matching paths will be returned. Since a connector may not guarantee that data is always there, this will not generate an error. For example if the path asks for A.B.C but only some instances of A contain B and only some instances of B contain C, the result skips the missing B and C entries.

In virtually all cases, widgets expect the data from a data path to result in strings or numbers. If your path does not evaluate to one of these types, the Splunk SOAR (On-premises) web interface will not show anything.

Do not use spaces in your key names.

Use custom views to render results in your app

Splunk SOAR (On-premises) lets app authors use a custom view by rendering the results of an action in a tabular format without writing a single line of rendering code. The author lists the data paths in the output section of an action, specifies the column names and order for each data path of the table view, and then Splunk SOAR renders the custom view in Splunk Mission Control.

Splunk SOAR (On-premises) also provides a JSON view. The JSON view can be a useful placeholder that supports contextual actions based on the contains specified in the output list of the action in the app JSON. The app author doesn't need to specify the column order or names for this view. The author only sets the type as JSON in the render dictionary.

In complex scenarios, such as in a detonate file, the tabular or JSON view might not be the most effective way to render action results. The data can display, but using a custom view is often simpler.

Example: Creating a custom view in the Splunk SOAR DNS app

The Splunk SOAR DNS App comes preinstalled on Splunk SOAR (On-premises).

The DNS App uses a custom view to render the results of the lookup domain action. The action results are separated into multiple tables. The first table displays key-value pairs. The second table shows IPs that the action returns. Contextual actions using contains are also implemented in custom views.

Implement a custom view

A custom view implementation uses the Django template framework. To implement a custom view, follow these steps:

1. Use the app JSON to select a custom view function in the **render** section of the action.
2. In a separate Python file, implement the view function specified in the app JSON.
3. The **view** function returns the Django template HTML file used to render the result context.
4. Use the Django template HTML file in the DNS App directory.
5. Package the view, template, and other code in the app installer. The results display on the **Investigation** page in Splunk SOAR (On-premises).

Use the render section of the app JSON

To configure a custom view function in the app JSON, follow these steps:

1. In the app JSON, go to the **render** section.
2. Select **type** as **custom**.
3. Select **view** as the module function. The **view** value is made up of two parts:
    1. The Python file name, `dns_view`.
    2. The function within the Python file, `display_ips`.

Every action can have its own view function and its own HTML template file.

The following code shows what the render section of the app JSON looks like:

```
"render": {
    "type": "custom",
    "width": 10,
```

```
    "height": 5,
    "view": "dns_view.display_ips",
    "title": "Lookup Domain"
    },
```

Using the view function

The view function creates context, similar to a Python dictionary, that the template code uses to render data. The following code shows the view function section of the DNS App:

```
def display_ips(provides, all_app_runs, context):

    context['results'] = results = []
    for summary, action_results in all_app_runs:
        for result in action_results:

            ctx_result = get_ctx_result(result)
            if (not ctx_result):
                continue
            results.append(ctx_result)
    # print context
    return 'display_ip.html'
```

The following table shows the parameters used in the view function.

Field	Description
provides	The name of the action that was run. For example, lookup domain.
all_app_runs	A list of tuples containing all the results of the action run. For example, `[({u'total_objects': 1, u'total_objects_successful': 1}, [<ActionResult>])]`
context	The context with which your template renders. You might change the context, but you're limited to JSON-serializable data types. Before your template is rendered, the respective Django model objects override the `container` and `app` keys. `{` `u'QS': {` `u'app_run': [u'3'],` `u'container': [u'']` `},` `u'container': 1,` `u'no_connection': False,` `u'app': 19,` `u'dark_title_logo': u'dns_876ab991-313e-48e7-bccd-e8c9650c239c/logo_phantom_dark.svg',` `u'google_maps_key': False,` `u'title_logo': u'dns_876ab991-313e-48e7-bccd-e8c9650c239c/logo_phantom.svg'` `}`
ActionResult	This field represents the results of the action run. You can use the following methods to access the data: • `get_param()`: Get the parameters used to call the action. For example, `{u'domain': u'google.com', u'type': u'A', u'context': {u'guid': u'0b8d526f-991a-4926-94d4-ca51f72b4302', u'artifact_id': 0, u'parent_action_run': []}}` • `get_summary()`: Get a summary of the results. For example, `{u'record_info': u'172.217.6.78', u'cannonical_name': u'google.com.', u'total_record_infos': 1}`

Field	Description
	• `get_data()`: Get the raw result data. For example, `[{u'record_info_objects': [{u'record_info': u'172.217.6.78'}], u'record_infos': [u'172.217.6.78']}]`

To use the view function, modify the following items:

- Change the returned HTML file to be specific to the custom view's action. In this example, it appears as display_ip.html.
- You can either modify the `get_ctx_result(...)` function that is called for every result object, or you can code your own function. The result object represents every `ActionResult` object that is added in the action handler, usually one per action. The `get_ctx_result(...)` function converts the result object into a context dictionary by performing the following steps:

1. Initialize the `ctx_result` dictionary.
2. Get the `param` from the result object and set it to the param key of `ctx_result`.
3. Get the `summary` from the result object and set it to the summary key of `ctx_result`.
4. Get the data from the result object and set it to the data key of `ctx_result`. The data is converted from a list to a dictionary.

The `get_ctx_result(...)` function defined in the dns_view.py file looks like the following:

```
def get_ctx_result(result):

    ctx_result = {}
    param = result.get_param()
    summary = result.get_summary()
    data = result.get_data()

    ctx_result['param'] = param

    if (data):
        ctx_result['data'] = data[0]

    if (summary):
        ctx_result['summary'] = summary

     return ctx_result
```

The function performs the following steps:

```
ctx_result['data'] = data[0]
```

You can add data in the action handler by using the ActionResult::add_data(...) API. This interface always keeps data as a list which makes rendering code in the template more straightforward. The get_ctx_result(...) function converts data from a list of one item to a dictionary.

Use the template file

The template file is an HTML file given to the context dictionary, which it parses using Django template language and generates code that is responsible for rendering the data. Search for Django's template language documentation for more information on how to access context variables and interact with them. In its most basic form, the template file consists of the following:

View an example of the template file.

```
{% extends 'widgets/widget_template.html' %}
{% load custom_template %}
â
{% block custom_title_prop %}{% if title_logo %}style="background-size: auto 60%; background-position: 50%;
background-repeat: no-repeat; background-image: url('/app_resource/{{ title_logo }}');"{% endif %}{%
endblock %}
{% block title1 %}{{ title1 }}{% endblock %}
{% block title2 %}{{ title2 }}{% endblock %}
{% block custom_tools %}
{% endblock %}
â
{% block widget_content %} <!-- Main Start Block -->
â
<!-- File: display_ip.html
  Copyright (c) 2016-2020 Splunk Inc.
â
  SPLUNK CONFIDENTIAL - Use or disclosure of this material in whole or in part
  without a valid written license from Splunk Inc. is PROHIBITED.
-->
â
<style>
â
â
.dns-display-ip a:hover {
  text-decoration:underline;
}
â
â
.dns-display-ip .wf-table-vertical {
  width: initial;
  font-size: 12px;
}
â
.dns-display-ip .wf-table-vertical td {
  padding: 5px;
  border: 1px solid;
}
â
.dns-display-ip .wf-table-horizontal {
  width: initial;
  border: 1px solid;
  font-size: 12px;
}
â
.dns-display-ip .wf-table-horizontal th {
  text-align: center;
  border: 1px solid;
  text-transform: uppercase;
  font-weight: normal;
  padding: 5px;
}
â
.dns-display-ip .wf-table-horizontal td {
  border: 1px solid;
  padding: 5px;
  padding-left: 4px;
}
â
.dns-display-ip .wf-h3-style {
```

```
  font-size : 20px
}
â
.dns-display-ip .wf-h4-style {
  font-size : 16px
}
â
.dns-display-ip .wf-h5-style {
  font-size : 14px
}
.dns-display-ip .wf-subheader-style {
  font-size : 12px
}
â
</style>
<div class="dns-display-ip" style="overflow: auto; width: 100%; height: 100%; padding-left:10px;
padding-right:10px"> <!-- Main Div -->
  {% for result in results %} <!-- loop for each result -->
<br>
â
<!------------------- For each Result ---------------------->
â
<h3 class="wf-h3-style">Info</h3>
<table class="wf-table-vertical">
{% if result.param.domain %}
<tr>
  <td><b>Domain</b></td>
  <td>
    <a href="javascript:;" onclick="context_menu(this, [{'contains': ['domain'], 'value': '{{
result.param.domain|escapejs }}' }], 0, {{ container.id }}, null, false);">
    {{ result.param.domain }}
     <span class="fa fa-caret-down" style="font-size: smaller;"></span>
    </a>
  </td>
</tr>
<tr>
  <td><b>Type</b></td>
  <td>
    {{ result.param.type }}
  </td>
</tr>
{% endif %}
</table>
â
<br>
â
    <!-- IPs -->
      {% if result.data.record_infos %}
      <table class="wf-table-horizontal">
        {% if result.param.type == 'A' or result.param.type == 'AAAA' %}
          <tr><th>IP</th></tr>
        {% else %}
          <tr><th>Record Info</th></tr>
        {% endif %}
        {% if result.param.type == 'A' or result.param.type == 'AAAA' %}
          {% for record_info in result.data.record_infos %}
          <tr>
            <td><a href="javascript:;" onclick="context_menu(this, [{'contains': ['ip', 'ipv6'], 'value':
'{{ record_info|escapejs }}' }], 0, {{ container.id }}, null, false);">
                {{ record_info }}
                 <span class="fa fa-caret-down" style="font-size: smaller;"></span>
            </a></td>
```

```
          </tr>
          {% endfor %}
        {% else %}
          {% for record_info in result.data.record_infos %}
            <tr>
              <td>{{ record_info }}</td>
            </tr>
          {% endfor %}
        {% endif %}
â
      </table>
      <br>
      {% else %}
      <p> No Record Info in results </p>
      {% endif %}
â
â
<!------------------- For each Result END ---------------------->
â
{% endfor %} <!-- loop for each result end -->
</div> <!-- Main Div -->
â
{% endblock %} <!-- Main Start Block -->
```

You can use the supplied display_ip.html template as the basis for your own custom view template. However, you must make the following changes:

1. Put any custom styles your app might require inside `<style></style>` tags and before the code that renders the actual results. Do not set any colors for text because they might not render properly.
2. Make the changes required inside the `div` tag commented with `Main Div`. Loop over the data you added to context in your view function. For example, if you added the results to context through context['results'], change the following {% for result in results %}...{% endfor %} in the template. For the context menu to work in your template, include the following tag in your template with the appropriate data:

 {{ record_info }}   For more information, see Use the contains parameter to configure contextual actions.

Modify the render code for each result

The DNS App renders two tables as shown in the following example.

Here is the code from the first table:

```
     <h3 class="wf-h3-style">Info</h3>
     <table class="wf-table-vertical">
     {% if result.param.domain %}
     <tr>
          <td><b>Domain</b></td>
          <td>
               <a href="#" onclick="context_menu(this, [{'contains': ['domain'], 'value': '{{
result.param.domain|escapejs }}' }], 0, {{ container.id }}, null, false); return false;">
     {{ result.param.domain }}

               </a>
          </td>
     </tr>
```

```
     <tr>
          <td><b>Type</b></td>
          <td>
               {{ result.param.type }}
     </td>
     </tr>
     {% endif %}
     </table>
```
The second table is made up of one column that lists all the IPs that the domain resolves in. The code for this table looks like this:

```
     <!-- IPs -->
          {% if result.data.ips %}
          <table class="wf-table-horizontal">
               <tr><th>IP</th></tr>
               {% for curr_ip in result.data.ips %}
              <tr>
                    <td><a href="#" onclick="context_menu(this, [{'contains': ['ip'], 'value': '{{
curr_ip|escapejs }}' }], 0, {{ container.id }}, null, false); return false;">
                         {{ curr_ip }}
                          <span class="fa fa-caret-down" style="font-size: smaller;"></span>
                    </a></td>
               </tr>
               {% endfor %}
          </table>
          <br>
          {% else %}
          <p> No IPs in results </p>
          {% endif %}
```
The following statements describe the tables :

- The first **if** statement only creates the table if **IPs** are present in the **data** dictionary
- Next is the **table** definition.
- The first row sets the header name **IP**.
- The other rows are added in a loop for the number of items in the **result.data.ips**.
- The anchor for the contextual actions for each IP are added in its own cell.

Use the app_resource endpoints to include files in custom views

If your custom views need to include other assets, the `app_resource` endpoints are a good way to retrieve files from an app by using HTTP. These endpoints can retrieve any files that belong to an app, including custom HTML files, the app logo, or other custom resources your app can define. You can then include these files in your custom view. The following endpoint shows what a general `app_resource` looks like. Make sure you include the app name, and the app's Globally Unique Identifier (GUID) when using the endpoint.

```
https://<instance_ip>/app_resource/<appname>_<appguid>/path/to/file
```
In the following example request, the endpoint is used to request the MaxMind app logo.

```
https://<instance_ip>/app_resource/maxmind_c566e153-3118-4033-abda-14dd9748c91a/logo_maxmind_dark.svg
```
Debug custom views

Your custom view is executed by an app interface module and all errors are recorded in the /var/log/phantom/app_interface.log file, unless your function was incorrectly defined or your template is invalid, in which

case it is logged to /var/log/phantom/wsgi.log.

Using Python logging to debug a custom view

You can use Python logging in your REST handler to debug a custom view. Any logging at level WARNING or higher is logged into the /var/log/phantom/app_interface.log file. Use the following code to define your logger:

```
import logging
```

logger = logging.getLogger(__name__)

After you define your logger, use the following code to log a message: `logger.warning('Example warning')`

Debug a ViewDoesNotExist exception

The ViewDoesNotExist exception occurs due to the following issues:

- The __init__.py file is missing from the app folder.
- The view file name doesn't match the name specified in the JSON file.
- The function name doesn't match the name specified in the JSON file.

Use REST handlers to allow external services to call into Splunk SOAR (On-premises)

REST handlers allow external services to call into Splunk SOAR (On-premises). When you create an app, you can use REST handlers in a variety of ways. Two particularly useful ways are for reparsing your data to push it into Splunk SOAR (On-premises) and for using OAuth with test connectivity. When reparsing your data, you can also use the REST Data Source app and supply a data transformation script. For more information see Use a Custom Script in the *REST API Reference for Splunk SOAR (On-premises) Manual*. The ability to write your own implementation is also provided, if you need more flexibility.

Using REST handlers

When you make an API request to Splunk SOAR (On-premises), your code is called with a Django request object and a list of URL parts. To use a REST handler, follow these steps:

1. Enter the URL you want to use in the REST handler after the prefix `rest/handler/[appname]_[appid]/[asset_name]`. You can find the URL by navigating to the main menu and clicking **Apps > Configure New Asset > Asset Settings** after your REST handler is configured and the app JSON code is updated. Splunk SOAR (On-premises) then calls into your function.
2. Parse your desired data from the request object and the URL fragments.
3. Parse the authentication header from the request object.
4. Create appropriate containers and artifacts through the Splunk SOAR REST API, using the parsed authentication header. For more information on creating containers and artifacts, see REST Containers and REST Artifacts in the *Splunk SOAR (On-premises) REST API Manual*.
5. Return your desired response.

REST handlers can't import any Splunk SOAR platform code. But REST handlers can import any dependencies defined in your app JSON, and any of Django's built-in modules, but calling anything that depends on django.setup()

fails.

Arguments

REST handlers receive the following arguments:

Name	Description
Request	This is a standard Django request object, but with a few exceptions. For example, this object doesn't have any attributes set by the middleware or application such as request.user, request.session, or request.site.
path_parts	This is a list of strings that contains all the URL parts after the REST handler prefix. For example, if you call the URL`https://your.phantom.url/rest/handler/[appname]_[appid]/[asset_name]/part1/part2/part3/`, the `path_parts` would be ["part1", "part2", part3"]. You can use this URL to help you create a URL structure for your REST handler.

Return values

You can return any of the Django HTTPResponse objects from your REST handler, but in most cases returning either HTTPResponse or JSONResponse, or raising HTTP404 is sufficient. For more information, search for HttpResponse objects on the Django website.

Example: ingestion

For the example_app_rest_handler.py file name, you must add your REST handler to the app JSON. In the following example, the correct value is "rest_handler": "example_app_rest_handler.handle_request".

```
import json
import requests
import logging

from django.http import HttpResponse, JsonResponse

from phantom_common.install_info import get_rest_base_url


logger = logging.getLogger(__name__)
REST_BASE_URL = get_rest_base_url()


def _get_auth_token_from_request(request):
    """Parse authentication information from request headers."""
    auth_token = request.META.get('HTTP_PH_AUTH_TOKEN')
    if not auth_token:
        raise Exception('Invalid request. Please use "ph-auth-token" to authenticate the request')

    return auth_token

def handle_request(request, path_parts):
    """Example rest_handler function."""
    data = json.loads(request.body)

    # Parse the received data and convert it to a dict that can be sent to the Splunk SOAR REST API
    # Here we're just gonna assume data['container'] already contains the correct data.
    container_data = data['container']

    # Headers to authenticate calls to phantom REST api
    headers = {'ph-auth-token': _get_auth_token_from_request(request)}
```

```
    # Build the correct url
    url = REST_BASE_URL + 'container/'

    # Send a request to the REST API
    response = requests.post(url, json=container_data, headers=headers)
    response_json = response.json()

    # Check if container was successfully created.
    if response_json.get('success', False) is True:
        container_id = response_json.get('id')
        return JsonResponse({'success': True, 'created_container_id': container_id})

    else:
        return HttpResponse(
            'Container creation failed. Response: {}'.format(response_json),
            status=400)
```

Debugging REST handlers

Your REST handler is executed by an app interface module and all errors are recorded in the /var/log/phantom/app_interface.log file, unless your function was incorrectly defined. In that case, the error is logged in /var/log/phantom/wsgi.log

To debug a REST handler, follow the guidance that best matches your situation:

Using Python logging to debug a REST handler

You can use Python logging in your REST handler for debugging purposes. Any logging at level WARNING or higher is logged into the /var/log/phantom/app_interface.log file. Use the following command to define your logger:

```
import logging
```

logger = logging.getLogger(__name__)

After you define your logger, use the following command to log a message: `logger.warning('Example warning')`

Debugging a ViewDoesNotExist exception

A ViewDoesNotExist exception is a common error. This error often occurs due to the following issues:

- The__init__.py file is missing from the app folder.
- The view file name doesn't match the name specified in the JSON file.
- The function name doesn't match the name specified in the JSON file.

Frequently asked questions

- How do I handle Python module dependencies for my app?

How do I handle Python module dependencies for my app?

You can manage dependencies on Python modules by packaging the required modules with the app, or by adding PIP dependencies in the app JSON.

When an action is executed, the platform adds the following directories to the `PYTHONPATH` environment:

- /opt/phantom/lib
- /opt/phantom/www
- /opt/phantom/apps/[app_install_directory]
- /opt/phantom/apps/[app_install_directory]/dependencies

Package required modules with the app

Place all required modules as part of the app TAR file. You can do this in the following ways:

- Install the modules into a sub directory of the app. PIP supports the `--target` command line switch that allows the modules to be installed at a specific location. Use this switch to install the modules into the app's subdirectory called `dependencies`. When the app TAR file is installed on the platform the modules will be part of the app code.
- Distribute the complete module, including the source and license file, in a sub directory of the app. In this case the app will need to append any folders in the `PYTHONPATH` it self.

The platform will install all the files present in the app TAR file in the app install directory. Be careful while packaging modules this way. Many Python modules are released under various licenses. Make sure the license allows for such redistribution. Packaging modules this way makes the app self-sufficient.

Add PIP dependencies in the app JSON

Specify app dependencies in the app JSON so that the platform tries to install the dependencies with the app. See Specifying pip dependencies for more information.

App Templates

Table Template

In order to define a table widget, set the render type to `table` and add a `template` key. This will contain an array of dictionaries. Each element of the array will represent one column of the table.

As an example, if a connector adds items like the following to the CommandResult as shown:

```
{
  "ip": "<ip address>",
  "container": {
    "filename": "<filename>",
    "pid": <process id>,
    "other data": ...,
}
```

And then sets `succeeded` to True or False in the CommandResult summary, then you can write the following template to generate a three column table.

```
{
  "name"    : "Connector Name",
  ...
  "commands": [
    {
      "provides": "list processes",
      ...
      "render": {
        "type": "table",
        "width": 4,
        "height": 5,
        "title": "My App Output"
        "menu_name": "My App"
      },
      "output": [
        {
          "data_path": "data..container.hash"
          "data_type": "string",
          "contains": ['hash'],
          "column_name": "File Hash",
          "column_order": 0,
        },
        {
          "data_path": "data..container.pid"
          "data_type": "numeric",
          "contains": ['pid'],
          "column_name": "Process ID",
          "column_order": 1,
        },
        {
          "data_path": "data..ip"
          "data_type": "string",
          "contains": ['ip'],
          "column_name": "IP",
          "column_order": 2,
        },
```

```
        {
          "data_path": "data_summary.succeeded"
          "data_type": "string",
          "contains":[],
          "column_name": "Some Column Name",
          "column_order": 3,
        }
      ]
    }
  ]
}
```

If a render type of table is specified, then the table view uses the following fields from the `output` key in order to render your results:

Field	Required?	Description
data_path	Required	Specifies the Data Path of this field. Data Paths are a method of indexing into the JSON in an abstract fashion that allows others who want to access that data to do so by specifying the appropriate path. The Data Path specified here must be populated by the connector code that creates and returns this field. This string value can start with `command_data`, `command_summary` or `summary` for accessing the CommandResult data or the summary data for the entire connector run. See Use data paths to present data to the Splunk SOAR web interface for more information about data paths.
data_type	Required	The type of variable. Supported types are: `string`, `password`, `numeric`, `boolean`.
contains	Optional	Specifies what kind of content this field contains. The types listed here tell the UI what kind of context sensitive actions can be run on this value when it is displayed. For example, an `ip geolocation` action is something that can be applied to any IP address that appears in a column which lists `ip` as a type. In the previous example, On the first column a user can run file based actions such as a `file reputation` lookup. On the second column they can run a `terminate process`. This is because there are apps that support those actions that take a `hash` and `pid` as input. No actions are available for the 3rd column.
column_name	Required	Specifies the name of this column within the rendered widget when it is displayed. This provides a convenient way to show data in Mission Control and when viewer action results in tabular form.
column_order	Required	Specifies the order of this column. Column ordering starts at 0.

Map Template

In order to define a map widget, set the render type to `map` and add a `template` key. This will be a single dictionary.

Defining a map widget

If a connector adds items like the following to the CommandResult:

```
{
  "latitude": 37.4487,
  "longitude": -122.1181,
  "address": "1.2.3.4",
  "description": "A nice place to live"
}
```

Then generate a map with map markers describing each location with the following template:

```
{
  "name"    : "Connector Name",
  ...
  "commands": [
    {
      "provides": "find locations",
      ...
      "render": {
        "type": "map",
        "width": 4,
        "height": 5,
        "title": "IP Geolocation"
      },
      "output": [
        {
          "data_path": "action_result.data.*.longitude",
          "data_type": "numeric",
          "map_info": "longitude"
        },
        {
          "data_path": "action_result.data.*.latitude",
          "data_type": "numeric",
          "map_info": "latitude"
        },
        {
          "data_path": "action_result.data.*.address",
          "data_type": "string",
          "contains": [ "ip" ],
          "map_info": "name"
        },
        {
          "data_path": "action_result.data.*.description",
          "data_type": "string",
          "map_info": "Description"
        }
      ]
    }
  ]
}
```

The `template` dictionary has the following members:

Member	Required?	Description
data_path	Required	Specifies the Data Path of this field. Data Paths are a method of indexing into the JSON in an abstract fashion that allows others who want to access that data to do so by specifying the appropriate path. The Data Path specified here must be populated by the connector code that creates and returns this field. This string value can start with `command_data`, `command_summary` or `summary` for accessing the CommandResult data or the summary data for the entire connector run. See Use data paths to present data to the Splunk SOAR web interface for more information about data paths.
data_type	Required	The type of variable. Supported types are: `string`, `password`, `numeric`, `boolean`.
contains	Optional	Specifies what kind of content this field contains. The types listed here tell the UI what kind of context sensitive actions can be run on this value when it is displayed. For example an `ip geolocation` action is something that can be applied to any IP address that appears in a column which lists `ip` as a type.

Member	Required?	Description
		In the previous example, on the first column you can run file based actions such as a `file reputation` lookup. On the second column, you can run a `terminate process`. This is because there are apps that support those Actions that take a `hash` and `pid` as input. No actions are available for the 3rd column.
map_info	Required	The parameters required to render a data point on the map. One `map_info` value must be provided for each of `longitude`, `latitude`, and `name`.

Inputting a Google API Key

To input a Google API Key, complete the following steps in Splunk Phantom:

1. From the main menu, select **Administration**.
2. Select *Administration Settings > Google Maps*.
3. From there, insert the API key.

Python

Platform installation for Python 3

The Splunk SOAR installation include a Python 3 runtime environment.

Python installation path

Only Python 3 is included in the platform installation.

- The path to Python 3 is `<SOAR_HOME>/usr/python39/bin/python3.9m`.
- For Splunk SOAR (Cloud) installations, `<SOAR_HOME>` is `/opt/phantom`.

A symlink to `<SOAR_HOME>/opt/phantom/usr/bin/python3` is included for convenience. Splunk SOAR (Cloud) uses this symlink to access Python.

Pip installation path

Pip 3 is included in the platform for installing Python 3 packages.

Pip 3 is included in `<SOAR_HOME>/usr/python36/bin/pip3`.

A symlink to pip 3, `<SOAR_HOME>/usr/bin/pip3` is included. Splunk SOAR (Cloud) uses this symlink to access pip 3.

App development script installation path

This entire section refers to scripts and CLI/SSH items which are not available in Splunk SOAR (Cloud).

The following scripts and commands are included in the platform for developing Python 2 and Python 3 apps:

- `create_tj.pyc`
- `create_output.pyc`
- `phenv compile_app` command

The script names are the same for each environment, the options are the same, and the outputs are the same.

Compatible Python 3 scripts

Scripts that are compatible with Python 3 are included in the following location:

- For standard installations, `<SOAR_HOME>` is `/opt/phantom`.
- For non-root installations, `<SOAR_HOME>` is configured by the customer.
- The path to scripts is `<SOAR_HOME>/bin/<script>`.

Run the scripts using `phenv python3` as follows:

```
[phantom@phantom phipinfoio]$ phenv python <SOAR_HOME>/bin/<script>.pyc
```

Convert apps from Python 2 to Python 3

This entire topic relies on tools which require SSH access to the SOAR instance. We need to decide what we're going to do for SOAR customers who need these tools. Remove this note before beta.

Convert your existing apps from Python 2 to Python 3.

Prerequisites

Your code is syntactically correct and you're able to successfully run the script for `<app>_connector.py` on the existing Python 2 app version. In the following example, `phantom` is the user, `phipinfoio` is the app directory, and `ipinfoio` is the app name:

```
[phantom@phantom phipinfoio]$ phenv python2.7 ipinfoio_connector.py
```

Run the 2to3 tool on the app

The 2to3 tool is located in `${PHANTOM_HOME}/bin`

1. Run the following command:
   `[phantom@phantom <app-dir>]$ phenv 2to3 <app-name>_connector.py`
   This will output the recommended changes to make the code Python 3 compatible.
2. Review the recommended changes and make sure they make sense before putting them in.
   Some suggestions are wrong or unnecessary. The following is an example of a necessary change:

```
-        print (json.dumps(json.loads(ret_val), indent=4))
+        print((json.dumps(json.loads(ret_val), indent=4)))
```

Make further changes that 2to3 missed

2to3 is not perfect. It misses things like certain default module name changes.

The best way to figure that out is to test your app. You can test from the UI or from the command line. Run the app, and make changes where it crashes until it works. See Compile and install.

Revise any d.iteritems() in the app .py file

Check if your code contains the following statement:

```
for k,v in d.iteritems():
```

If so, replace `d.iteritems()` with `six.iteritems(d)`.

Six is included, but you need to import it at the top of your file.

```
import six
...
six.iteritems(d)
...
```

Update the app .json file

Update the `python_version` and `app_version` keys in the app JSON file for the changes to take effect.

Update python_version key

Update the `python_version` key in the app JSON to the string '3'.

```
{
  "app_config_render": null,
  "product_version_regex": ".*",
  "python_version": "3",
  "uber_view": null,
  "disabled": false,
...
}
```

To compile using Python 3, the `phenv compile_app` command expects the `python_version` key in the app JSON to be set to 3. Other acceptable values are strings of 3.9. Use 3 in preparation for any potential platform migrations beyond python3.9. For example, if you don't use 3, you have to upgrade your app if Python is upgraded on the platform.

If you only have .py files, the compile script will use the version that you specify in your app .json file. That generates the .pyc files that the interpreter uses.

See App development script installation path for script usage.

Update app_version key

Update the `app_version` key to a higher version number.

```
{
...
 "app_version": "1.0.9",
  "type": "reputation",
  "product_name": "AbuseIPDB",
...
}
```

The `app_version` key accepts any update, such as from 1.0.0 to 1.0.1 or from 1.0.0 to 2.0.0. As long as the number is higher, the UI understands that it's an upgrade.

It is possible to switch back and forth in the UI between the app version that uses Python 2 & the version that uses Python 3. If you downgrade and go back to using the Python 2 version in the UI, it automatically runs the Python 2 code.

Dependencies for older versions

In the case of converting an app that is compatible with current and older versions of Splunk Phantom, there are multiple dependencies to consider.

If your app has custom views or its own REST handler, it needs to be compatible with Python 2 and ship with the source until a future release of Splunk Phantom when the entire platform is fully python3 compatible.

If your converted app has a dependency with functionally-equivalent versions, one of which is only available for Python 2 and the other for Python 3, then you might need to define `pip3_dependencies` in the app JSON alongside

`pip_dependencies` so that the platform will install the different versions as needed. You only need to define `pip3_dependencies` if your app has a different set of dependencies when running in Python 3 than when running in Python 2. Otherwise, the platform will accept a Python 3 app that only defines `pip_dependencies` if the dependencies are the same.

Compile and reinstall

After completing all the steps on this page, verify if your code is syntactically correct. Verify it by running the compatible Python 3 equivalent script for `<app>_connector.py`, such as in the following example:

```
[phantom@phantom phipinfoio]$ phenv python3 ipinfoio_connector.pyc
```

Use the compatible Python 3 script to reinstall the app in the app directory. This compiles and installs it in the UI and lets you test the actions, such as in the following example:

```
[phantom@phantom phipinfoio]$ phenv python /opt/phantom/bin/compile_app.pyc -i
```

See Compatible Python 3 scripts for more information about Python 3 scripts.

See your new app version in the UI

Both app versions are available in the Splunk Phantom UI.

1. Navigate to the **Main Menu**.
2. Select **Apps**.
3. Scroll to find the app or search for it by name.
4. Use the drop-down menu to to see the version numbers.