Splunk® SOAR (On-premises)
Python Playbook API Reference for Splunk SOAR (On-premises) 5.4.0

Generated: 10/27/2022 8:33 pm

Table of Contents

Overview

About Splunk SOAR (On-premises) playbook automation APIs

The Splunk SOAR (On-premises) playbook automation API allows security operations teams to develop detailed automation strategies. Playbooks serve many purposes, ranging from automating small investigative tasks that can speed up analysis to large-scale responses to a security breach. The Splunk SOAR (On-premises) playbook automation APIs are supported to leverage the capabilities of the platform.

The Splunk SOAR (On-premises) playbook automation APIs operates using containers, artifacts, datapaths, apps, and assets. To learn more about containers, artifacts, datapaths, apps, and assets, see the following sections:

- Containers are the top-level data structure the playbook automation APIs operate on. A container is a composite object that consists of one or more artifacts that can be automated against. To learn more about containers and artifacts, see Understanding containers and Understanding artifacts.
- To learn more about the datapaths used by the Splunk SOAR (On-premises) playbook automation APIs, see Understanding datapaths.
- Apps are included and shipped with Splunk SOAR (On-premises) and provide actions that are used by the Splunk SOAR (On-premises) playbooks. Assets are instances of apps configured by a Splunk SOAR (On-premises) admin. To learn more about apps and assets, see Understanding apps and assets.

The automation APIs are made up of the following groups: the playbook automation API, the container automation API, the data management automation API, the data access automation API, the session automation API, the vault automation API, and the network automation API. For more information about the automation APIs, see Automation API.

See also

For more information about how to leverage the Splunk SOAR (On-premises) platform to perform automation, see the documentation.

You want to do this	Documentation
Create, update, and selectively remove objects from the system.	See REST API Reference for Splunk SOAR (On-premises).
Create a playbook to automate your workflows.	See Build Playbooks with the Playbook Editor.
Examples of using some of the key automation API functions.	See the Python Playbook Tutorial for Splunk SOAR (On-premises) manual.

Understanding containers

Containers are the top-level data structure that Splunk SOAR (On-premises) playbook APIs operate on. Every container is a structured JSON object which can nest more arbitrary JSON objects, that represent artifacts. A container is the top-level object against which automation is run.

Assign a label to a container to dictate the kind of content it contains. This label defines how the respective elements are managed within the platform and where they are organized. Assign this label during the ingest phase and in the ingest configuration when you configure an asset as a data source.

Following this model, you might label containers imported from a SIEM as "Incidents". Or you might label containers imported from a vulnerability management product as "Vulnerabilities", or containers imported from an IP intelligence source as "Intelligence". For each label that the system ingests, a new top-level menu item appears within the top-level

product navigation to allow you to navigate to the list of respective containers for that label. In addition, playbooks, the mechanism by which automated actions are run on a container, are container-specific and run only on containers that match their label.

This architecture allows for arbitrary data to be imported and run across the security operations domain, beyond the management of security incidents alone.

Data can be imported from various sources and can be structured or unstructured. Even in the case of structured data, information might be categorized, classified, named, and represented in incompatible and disparate formats. In either case, the data has to be normalized to be accessible and actionable by the platform. The Splunk SOAR (On-premises) platform uses apps that support ingest and interface with these assets. These apps provide the necessary functionality to map the raw data format from the source to a standard Common Event Format (CEF) schema, if applicable.

CEF is an open log management standard that improves the interoperability of security-related information from different network and security devices and applications. After the data is normalized into CEF format, automation can leverage accessing various attributes without any ambiguity. Splunk SOAR (On-premises) makes both the original data in its native format and normalized format available because there is an information fidelity and application-specific detail in the raw data that might not be well-represented in CEF format.

Container schema

The contents of the container header and associated container data are exposed to the Splunk SOAR (On-premises) platform as JSON objects. Playbooks operate on these elements in order to make decisions and apply logic. The following code shows an example of the container schema, and the table immediately after the code defines the fields present in the code:

```
{
  "id": 107,
  "version": "1",
  "label": "incident",
  "name": "my_test_incident",
  "source_data_identifier": "64c2a9a4-d6ef-4da8-ad6f-982d785f14b2",
  "description": "this is my test incident",
  "status": "open",
  "sensitivity": "amber",
  "severity": "medium",
  "create_time": "2016-01-16 07:18:46.631897+00",
  "start_time": "2016-01-16 07:18:46.636966+00",
  "end_time": "",
  "due_time": "2016-01-16 19:18:00+00",
  "close_time": "",
  "kill_chain": "",
  "owner": "admin",
  "hash": "52d277ed6eba51d86190cd72405df749",
  "tags": [""],
  "asset_name": "",
  "artifact_update_time": "2016-01-16 07:18:46.631875+00",
  "container_update_time": "2016-01-16 07:19:12.359376+00",
  "ingest_app_id": "",
  "data": {},
  "artifact_count": 8
}
```

Field	Description
id	A unique identifier for the incident, generated by the Splunk SOAR (On-premises) platform.

Field	Description
version	The version of this schema, for schema migration purposes.
label	The label as specified in the ingest asset. When you configure an ingestion asset, you can define a label for the containers ingested from that asset. For example, you might define "Incident" from a Splunk asset or "Email" from an IMAP asset.
name	The name of the item, as found in the ingest or source application and incident name in a SIEM.
source_data_identifier	The identifier of the container object as found in the ingestion source. An incident in a SIEM can have an identifier of its own that might be passed on to Splunk SOAR (On-premises) as part of the ingestion. In the absence of any source data identifier provided, a GUID is generated and provided.
description	The description for the container as found in the ingest source.
status	The status of this container. For example, you can define a status as new, open, closed, or as a custom status defined by an administrator.
sensitivity	The sensitivity of this container such as red, amber, green, or white.
severity	The severity of this container. For example, you can define severity as medium, high, or as a custom severity defined by an administrator.
create_time	The timestamp of when this container was created in Splunk SOAR (On-premises).
start_time	The timestamp of when activity related to this container was first seen. This is also the time when the first artifact was created for this container. As artifacts are added to the container, the start time might change if an older artifact for that incident is added to the container.
end_time	The timestamp of when activity related to this container was last seen. This is also the time when the last artifact was created for this container. As artifacts are added to the container, start time may change if a later artifact for that incident is added to the container.
due_time	The timestamp of when the SLA for this container expires and it is considered to be in breach of its SLA. The SLAs for the container are either set by the user or default determined by the platform depending on the severity and the Event Settings. You can define default SLAs for container severity from **Main Menu > Administration > Event Settings > Response**.
close_time	The timestamp of when this container was closed or resolved by the user or playbook.
kill_chain	If the ingestion source and app provide kill-chain information about the incident, it's stored in this field.
owner	The user who currently owns the incident. Administrators can assign the container to any user in the system.
hash	This is the hash of the container data as ingested and is used to avoid duplicate containers being added to the system tags assigned to a container.
asset	The ID of the asset from which the container was ingested. If the user created this incident, this field does not contain a value.
asset_name	The name of the asset from which the container was ingested. If the user created this incident, this field does not contain a value.
artifact_update_time	The time when the artifact was last added to the container.
container_update_time	The time when the container was last updated. This includes adding an artifact or changing any state or field of the container.
data	This is a dictionary of the raw container data as seen in the ingestion asset or application. This is the data that is parsed to populate the artifacts and its CEF fields.
artifact_count	This is the count of total artifacts that belong to this container.

All containers imported in the system start with an open state and can eventually be closed by playbooks or the user. A

closed state implies that the Splunk SOAR (On-premises) platform no longer runs playbooks automatically, but the container continues to be updated with new artifacts if any are found and the users can take actions manually or run playbooks manually.

You can always re-open the container through the user interface.

There are some variations in fields returned when the container is queried or retrieved from a REST API versus the container object passed in playbook APIs.

Understanding artifacts

Artifacts are JSON objects that are stored in a container. Artifacts are objects that are associated with a container and serve as corroboration or evidence related to the container. Much like the container schema, the artifact schema has a common header that can be operated on, and also contains a Common Event Format (CEF) body and raw data body to store elements that can be accessed by Splunk SOAR (On-premises) playbooks as shown in the following code. The fields in the code are defined in the table immediately following the code:

```
{
  "id": 1,
  "version": 1,
  "name": "test",
  "label": "event",
  "source_data_identifier": "140a7ae0-9da5-4ee2-b06c-64faa313e94a",
  "create_time": "2016-01-18T19:26:39.053087Z",
  "start_time": "2016-01-18T19:26:39.058797Z",
  "end_time": null,
  "severity": "low",
  "type": null,
  "kill_chain": null,
  "hash": "7a61100894c1eb24a59c67ce245d2d8c",
  "cef": {
    "sourceAddress": "1.1.1.1"
  },
  "container": 1,
  "description": null,
  "tags": [""],
  "data": {}
}
```

Field	Description
id	A unique identifier for the artifact, generated by the Splunk SOAR (On-premises) platform.
version	The version of this schema.
name	A name of the artifact as identified by the ingestion app from the data source.
label	The label as identified by the app that is ingesting the data. Labels can be anything that is found to be the label of the data or event in the ingestion data source. For example, labels can be event, FWAlert, AVAlert, to name a few.
source_data_identifier	The identifier of the artifact as found in the ingestion data source.
create_time	The timestamp of when this artifact was created in Splunk SOAR (On-premises).
start_time	The timestamp of when this artifact was first seen. This timestamp typically coincides with when the artifact was initially detected or produced by the device that generated it.

Field	Description
end_time	The timestamp of this artifact as last seen in the ingestion data source.
severity	The severity of this artifact. For example, medium, high, or a custom severity created by an administrator.
type	The type of artifact. The type is used to identify the origin of this artifact, such as "network" or "host".
kill_chain	The kill-chain value as specified by the ingestion app and data source.
hash	The hash of the contents of the artifact. The hash is used by the platform to avoid saving duplicate artifacts for the same container.
cef	A normalized representation of the data mapped to each field's representative CEF key.
container	The ID of the container that contains this artifact.
tags	The list of tags assigned to this artifact.
data	The raw representation of the data.

Artifact deduplication

As a case proceeds, users and automation can add artifacts of interest. Splunk Phantom attempts to prevent duplicate artifacts from appearing in the system. However, inconsistencies occur when using a mix of input sources. For example, one user can't add an artifact twice, but different users can add the same artifact. When adding an artifact to a case with the Phantom app's "create container" action, adding the same artifact with the playbook api `phantom.add_artifact` will add a duplicate, but running `phantom.add_artifact` again for the artifact will be rejected.

A hash of the entire artifact body is taken to determine if it is a duplicate. Deduplication is determined by the hash, and the hash is based on the body of the POST request. A component of the artifact is the source ID, which is different depending on the how the artifact was created. The `source_data_identifier` is important in managing artifact duplication. If the the POSTed artifact doesn't have an SDI, the artifact is never deduplicated. Only if you add the same artifact twice, and they both have the same SDI, will deduplication happen. Deduplication doesn't trigger ingestion, even if "run_automation" is set to true on the artifact.

The hash is never changed after the fact, but it is sometimes altered intentionally to prevent collisions in certain scenarios such as cloning an artifact.

See also

- add_artifact in the *Python Playbook API Reference for Splunk Phantom*.
- Add artifacts from a container to a case in *Use Splunk Phantom*.

Understanding datapaths

The Splunk SOAR (On-premises) Playbook API uses a variety of datapaths. Use the following table to learn more about some of the important datapaths used in Splunk SOAR (On-premises), including their classification, description, and which APIs support them:

Datapath	Classification	Description	API support
artifact:	unnamed artifact	Refers to a field in the artifact. Begins with `artifact:` Example: `artifact:*.cef.bytesIn`	condition

Datapath	Classification	Description	API support
action-name:artifact:	named artifact	Refers to a field in the artifact that uses one or more Common Event Format (CEF) fields as an input parameter of an action. In the phantom.act() API, the auto-generated code uses the hidden parameter `context` that helps identify which artifact is used to run the action. The `context` parameter attaches a reference to the artifact that can then later be retrieved from the results. Each instance of `context` has a unique GUID and an optional `parent_action`. For example, `geolocate_ip_1:artifact` implies an `artifact_id` whose field is used as an input parameter for a particular action. Example: `geolocate_ip_1:artifact:*.cef.bytesIn`	condition
filtered_artifact:	filtered artifact	Refers to a field in the artifact that was filtered from a previous decision block. Example: `filtered_artifact:*.cef.bytesIn`	condition, collect2
action_result	action result	Refers to a field in the action result object. The condition API expects the `action_results` parameter to provide the action result list object. This object is the same object that is passed into the function by the `results` parameter or obtained by the `phantom.get_action_results()` API. Example: `action_result.parameter.ip`	condition, collect2
action_name:action_result.	action result	Refers to a field in the action result object of an action called with a `name` parameter. This field can be any of the actions that are run in the path that leads to this action. Example: `geolocate_ip_1:action_result.parameter.ip`	condition, collect2
action_name:filtered-action_result.	filtered action result	Refers to a field in the action result object of a named action that has been filtered. The phantom.condition() statement returns `action_result` objects or artifact IDs that match the condition. The field requires that the filtered results be provided as the parameters. Example: `geolocate_ip_1:filtered-action_result.parameter.ip`	condition, collect2
custom_function_result	custom function result	Refers to a field in the custom function result object. The condition API expects the `custom_function_result` parameter to provide the action result list object. This is the same object that is passed into the function through the `results` parameter or obtained from the phantom.get_action_results() API. Example: `custom_function_result.success`	condition, collect2
named_custom_function_result	named custom function result	Refers to a field in the `named_custom_function_result` object of a custom function called with a `name` parameter. This field can be any of the custom functions that may have been executed in the path that leads to this action. Example: `normalize_ip_1:custom_function_result.success`	condition, collect2
named_filtered_custom_function_result	filtered custom function result	Refers to a field in the `named_filtered_custom_function_result` object of a named custom function that has been filtered. A phantom.condition() statement returns those `custom_function_result` objects or artifact IDs that match the condition. When this type of datapath is used, it requires that the filtered results be provided as the parameter. Example: `normalize_ip_1:filtered-custom_function_result.success`	condition, collect2

Datapath	Classification	Description	API support
legacy_custom_function_result	legacy custom function result	References legacy, playbook specific, and custom functions by name. Example: `my_custom_function:custom_function:ip`	condition, collect2
named_artifact	named artifact	Reference artifacts by action name. This datapath is valid only if the artifact is used as an input to the action. Example: `geolocate_ip_1:artifact:*.cef.bytesIn`	condition, collect2
named_playbook_result_action	named playbook action result	Looks up the playbook by name, and then finds the action. Example: `pb1:playbook_results:geo1:action_result.parameter.ip`	condition, collect2
named_playbook_custom_function	named playbook custom function result	Datapaths of this classification reference a custom function result from a sub-playbook. Example: `pb1:playbook_results:geo1:custom_function_result.success`	condition, collect2
named_playbook_result_header	named playbook result	Specifies the named playbook result header. Once you have the result header, the datapath specifies the named playbook, and then indicates which custom function to look up. Example: `pb1:playbook_results:geo1:message`	condition, collect2
formatted_data	formatted data	This datapath ends with `:formatted_data`. The name of the format block is used to look up the formatted data. Example: `my_key:formatted_data`	condition, collect2
result_header	result header	Result headers are top-level keys that come out of the datapath results. Result headers can be named or unnamed. You can also get result headers from actions, custom functions, child playbooks or action or child playbook results where the results are actions or custom functions. Example: `action_1:message`	condition, collect2
none	none	This datapath is a null value. Example: `none`	condition
string	string	This datapath is any string value. Use the `condition` API to compare dynamic values to a constant value. If `myaction:status=="success"`, then you can pass in the string `"success"`. When strings are entered into the VPE configuration panel, any quotes that you use are considered string characters. Example: `192.0.2.0`	condition
numeric	number	This datapath is represented by any numbers. Example: `145.1833`	condition

Datapaths that point to a list of values

When you are publishing a list as the output of a custom function, it is best to produce a list of dictionaries so that a simple datapath can be used to index into the list or iterate over all items in the list. For example, the following custom function produces four numbers as output:

```
def primes_under_ten():
    import json
    from phantom.rules import debug

    # Custom code
    return [{"result": number} for number in numbers]
```

Because the output is returned as a list of dictionaries, you can use the following datapath can be used to see the entire list:

```
primes_under_ten_1:custom_function_result.data.*.number
```

Literal values in condition and collect2

Literal values are treated differently in the `condition` and `collect2` APIs. When calling the`condition` API, literal values are treated as such. But, when you call `collect2`, literal values are treated as result headers. Result headers correspond to top-level keys in the action results object.

You can remove any ambiguity by prefacing your result header with the action, custom function, or child playbook. For example, instead of using the data path `summary` use the data path `geolocate_ip_1:summary`.

The following example shows an action result. In this action result, the keys that are considered result headers are `status`, `data`, `message`, `parameter`, and `summary`:


```
[
    {
        "asset_id": 63,
        "status": "success",
        "name": "geolocate_ip_1",
        "app": "MaxMind",
        "action_results": [
            {
                "status": "success",
                "data": [
                    {
                        "state_name": "Victoria",
                        "latitude": -37.7,
                        "country_iso_code": "AU",
                        "time_zone": "Australia/Melbourne",
                        "longitude": 145.1833,
                        "state_iso_code": "VIC",
                        "city_name": "Research",
                        "country_name": "Australia",
                        "continent_name": "Oceania",
                        "postal_code": "3095"
                    }
                ],
                "message": "City: Research, State: VIC,
                            Country: Australia",
                "parameter": {
                    "ip": "1.1.1.1",
                    "context": {
                        "guid": "f42fd73f-...-8194aaa9bc11",
                        "artifact_id": 0,
                        "parent_action_run": []
                    }
                },
                "summary": {
                    "city": "Research",
```

```
                    "state": "VIC",
                    "country": "Australia"
                }
            }
        ],
        "app_id": 83,
        "summary": {
            "total_objects": 1,
            "total_objects_successful": 1
        },
        "asset": "maxmind",
        "action": "geolocate ip",
        "message": "'geolocate_ip_1' on asset 'maxmind': 1
                    action succeeded. (1)For Parameter:
                    {\"ip\":\"1.1.1.1\"} Message: \"City:
                    Research, State: VIC, Country: Australia\"",
        "app_run_id": 51,
        "action_run_id": 11
    }
]
```

Understanding apps and assets

Apps connect Splunk SOAR (On-premises) to third party services and provide actions that are used by playbooks. Assets are instances of apps configured by a Splunk SOAR (On-premises) admin. For more information, see About Splunk SOAR (On-premises) in the *Use Splunk SOAR (On-premises)* manual.

Apps and assets can be dynamically added to the system at any point in time through the user interface. Additionally, assets can also be added and managed through the REST API. As new apps are added, you are expected to define an asset on which the app can run an action. Apps are written for a specific product and each action of an app can indicate a regular expression for a version of the product that it supports. Assets also specify the product and version of the asset.

When Splunk SOAR (On-premises) runs an action, the platform intelligently matches the actions and assets for the corresponding product and version. Actions are run only on assets whose product and version match the app product and version. For example, the application can specify a product like Virus Total, and the action file reputation can specify a version, such as EQ(*), which implies that the action can run on all versions of Virus Total. Different apps for different products of the same type can provide the same action. For example, apps for Cisco ASA Firewall and Palo Alto Networks Firewall might both support the block IP action. When the action is run, you have the option to run either the action to block an IP on specific assets or on all of the assets where there is a matching app that supports that action. For more information, see the *Develop Apps for Splunk SOAR (On-premises)* manual.

Some assets can also provide the ability to ingest imported data to be imported. Data sources, such as a SIEM, can be configured for ingestion. For example, you can configure QRadar as an asset that also has ingestion configuration for Offenses, a term used by QRadar for incidents, to be imported into the Splunk SOAR (On-premises) platform. You can also use an asset ingestion configuration to define the basic properties of the containers, like the label or polling interval. For example, QRadar ingestion configuration can specify that all offenses imported from QRadar be labeled as incidents. The QRadar app provides the implementation to interface with the QRadar asset and map information in events on the QRadar system into CEF format as it is imported into Splunk SOAR (On-premises).

Understanding callbacks

In the context of Splunk SOAR (On-premises) playbooks, callbacks are Python callables that can accept the keyword arguments described in this section. For more information see https://docs.python.org/3.6/library/functions.html#callable

Callbacks can be used in the following two ways:

1. Through playbook code. For example, the code generated by a decision block will only call the callback if the return value of the call to phantom.decision is 'True'.
2. Through the automation engine. There are three playbook APIs that accept a callback keyword argument: phantom.act, phantom.custom_function, and phantom.playbook. When you pass a callable to these APIs, that callback will be invoked from within the automation engine once the action, custom function, or child playbook completes execution.

Callback functions can be invoked with any of the following keyword arguments: action, success, container, results, handle, filtered artifacts, filtered results, and custom function, depending on what the preceding block provides. Each of these parameters is discussed in greater detail in the following table.

Parameter	Description

action	This is a JSON object that specifies the action name and the action run ID. The action run ID uniquely identifies that instance of action execution and can be used to retrieve the details of the action execution. The following is an example of an action parameter from a callback:
	
	`{`
	`"action_run_id": 205,`
	`"action_name": "whois ip",`
	`"action" : "whois ip",`
	`"name": "my_whois_ip_action"`
	`}`
	The 'action' field replaces the action_name field. The action_name field will be removed in a future release.

successEither 'True' or 'False'. An action is considered failed only if the action has failed on all assets and with all specified parameters. If any part of the action succeeds, it is not considered failed. Use the utility functions like `parse_errors()` or `parse_success()` to get a flat listing of all errors and success. These utility functions parse the results to give the user a different view of the overall results.containerThis is the container JSON object. Here is an example of container parameter from a callback:

```
{
    "sensitivity": "amber",
    "create_time": "2016-01-14 18:25:55.921199+00",
   "owner": "admin",
    "id": 7,
    "close_time": "",
    "severity": "medium",
    "label": "incident",
    "due_time": "2016-01-15 06:24:00+00",
    "version": "1",
    "current_rule_run_id": 1,
    "status": "open",
    "owner_name": "",
    "hash": "093d1d4d22cab1c5931bbfb1b16ce12c",
    "description": "this is my test incident",
    "tags": ["red-network"],
    "start_time": "2016-01-14 18:25:55.926468+00",
    "asset_name": "",
    "artifact_update_time": "2016-01-14 18:26:33.55643+00",
    "container_update_time": "2016-01-14 18:28:43.859814+00",
    "kill_chain": "",
    "name": "test",
    "ingest_app_id": "",
    "source_data_identifier":
        "48e4ab9c-2ec1-44a5-9d05-4e83bec05f87",
    "end_time": "",
    "artifact_count": 1
```

```
}
```
resultsThis is a JSON object that has the full details and status of the complete action on all assets for each parameter. Here is an example where the geolocate ip action is run:

```
parameters = []
parameters.append({ "ip" : '1.1.1.1' })
phantom.act('geolocate ip', parameters=parameters,
    assets=["maxmind"], callback=geolocate_ip_cb,
    name='my_geolocate_action')
```

This simple action can result in various execution strategies and outcomes, depending on how the system is configured. In this simple form, one app supports the geolocate IP action and there is one 'Maxmind' asset that is configured which results in one IP being queried once on one asset. In a more complex example, if there are two Apps, both of which support file reputation, then this one simple action will result in a file hash being queried on both of the assets.

The single action file reputation will result in the hash being queried twice, once on each asset. The two queries still constitute one action, so the callback 'file_reputation_cb' will be called once when both the queries have completed. The parameters in the example is a list of dictionaries and contains one IP. But, in one action call the user can specify many different IP addresses.

The `results` JSON object provides full visibility into the execution of the action on all matching assets using all matching apps for all specified parameters.

Here are the results of the geolocate ip action.


```
[
    {
        "asset_id": 63,
        "status": "success",
        "name": "geolocate_ip_1",
        "app": "MaxMind",
        "action_results": [
            {
                "status": "success",
                "data": [
                    {
                        "state_name": "Victoria",
                        "latitude": -37.7,
                        "country_iso_code": "AU",
                        "time_zone": "Australia/Melbourne",
                        "longitude": 145.1833,
                        "state_iso_code": "VIC",
                        "city_name": "Research",
                        "country_name": "Australia",
                        "continent_name": "Oceania",
                        "postal_code": "3095"
                    }
                ],
                "message": "City: Research, State: VIC,
                            Country: Australia",
                "parameter": {
                    "ip": "1.1.1.1",
                    "context": {
                        "guid": "f42fd73f-...-8194aaa9bc11",
                        "artifact_id": 0,
                        "parent_action_run": []
                    }
                },
                "summary": {
```

```
                        "city": "Research",
                        "state": "VIC",
                        "country": "Australia"
                    }
                }
            ],
            "app_id": 83,
            "summary": {
                "total_objects": 1,
                "total_objects_successful": 1
            },
            "asset": "maxmind",
            "action": "geolocate ip",
            "message": "'geolocate_ip_1' on asset 'maxmind': 1
                        action succeeded. (1)For Parameter:
                        {\"ip\":\"1.1.1.1\"} Message: \"City:
                        Research, State: VIC, Country: Australia\"",
            "app_run_id": 51,
            "action_run_id": 11
        }
    ]
```

handleA string object that was specified in the action phantom.act() call for passing data between action and callbacks.filtered_artifactsRefers to a field in the artifact that was filtered from a previous decision block. For example, `filtered_artifact:*.cef.bytesIn`.filtered_resultsRefers to a field in the action result object of a named action that has been filtered. The phantom.condition() statement returns those action_result objects or artifact IDs that match the condition. Requires that the filtered results be provided as the parameters. For example, `geolocate_ip_1:filtered-action_result.parameter.ip`.custom_functionUse custom_function to call a custom function from a playbook.

The simplest callback looks like this:

```
def do_nothing(**kwargs):
    pass
```

This callback can accept all of the keyword arguments that are passed to it, but it doesn't do so. This callback function does not callback into any other function so playbook execution along this branch ends when this callback is invoked.

The following is an example of a callback that makes a debug call and then calls another callback. This callback isn't very useful in a real playbook, but it demonstrates how callbacks work.

```
def call_a_friend(**kwargs):
    phantom.debug('Invoking geolocate_ip_1')
    geolocate_ip_1(**kwargs)
```

Convert playbooks or custom functions from Python 2 to Python 3

Python 2 reached end of life status on January 1, 2020. The Splunk SOAR (On-premises) platform uses Python 3.

> Splunk Phantom playbooks written in Python 2.7 should be converted to Python 3 for use with Splunk SOAR, using an on-premises deployment of Splunk Phantom. Users who do not have an on-premises deployment of Splunk Phantom can download and install the free Community edition. See https://www.splunk.com/en_us/software/splunk-security-orchestration-and-automation.html.

Many Splunk Phantom deployments have significant numbers of playbooks and custom functions which were developed in Python 2 that need to be updated to Python 3. Splunk Phantom 4.10 and later releases contains two phenv commands to help you convert your custom functions and playbooks from Python 2 to Python 3.

- `customfunctions_to_py3`
- `playbooks_to_py3`

Best Practices for converting Splunk Phantom playbooks and custom functions from Python 2 to Python 3

Converting custom functions or playbooks to Python 3 requires some planning and careful work on both ends of the conversion process. Here are several best practices for doing the conversions.

- Create a full backup of your Splunk Phantom deployment. You will need this backup in order to restore your git repositories and their contents in the event that there is a problem during the conversion.
- Test each custom function and playbook before converting them. You must make sure the custom function or playbook is working correctly before proceeding. Converting incomplete, inoperable, or broken custom functions and playbooks can have unforeseen consequences.
- You should perform a dry run of each conversion using the `--dry_run` and `--verbosity 3` options of the `customfunctions_to_py3` and `playbooks_to_py3` phenv commands. Pay close attention to the output from these commands. They provide helpful guidance on potential pitfalls during the conversion process.
- Test each playbook or custom function after converting them before using the converted playbooks in production environments.
- Convert custom functions separately from converting playbooks.
- Playbooks and custom functions that have been converted to Python 3 will be new copies, with the `--suffix` appended to their names. These new copies need to be be linked to the relevant custom functions, playbooks, and any sub-playbooks after conversion by a Splunk Phantom administrator. Edit the relevant playbooks in the Visual Playbook Editor to update any blocks that reference an incorrect name.

customfunctions_to_py3 options and examples

These are the options available when using the customfunctions_to_py3 tool.

Argument	Required?	Description
repo/ repo2/playbook_name ...	Yes	A list of repositories or custom_functions to convert to Python 3. If repositories are listed, the tool will convert every custom_function in the given repositories. Only provide the repository's name, not the full path to the repository.
repo	Yes	Output repository for converted custom_functions.
-h, --help	No	Show the help message and exit the tool.
-d, --dry_run	No	Use this option to see which custom_functions would be converted with the tool, without doing the conversion.
-s <SUFFIX>, --suffix <SUFFIX>	No	Specify a suffix to be added to the names of converted custom_functions as part of the conversion process. If no suffix is specified the default is "_py3".
-v {0,1,2,3}, --verbosity {0,1,2,3}	No	Verbosity level: * 0 = minimal output * 1 = normal output * 2 = verbose output * 3 = very verbose output If you set --verbosity 3 you may see debug messages like this: Unsupported dict value of type: <class '_ast.NameConstant'>. Ignoring children and returning type./> These messages can be safely ignored. They are only informational messages from the code parser, and do not indicate a problem with the custom function.

Command examples

- Convert a single custom function from the repository production and output it to the repository local:

```
phenv customfunctions_to_py3 production/my_example_custom_function local
```

Example output:

```
Converted custom_function "my_example_custom_function"

Successfully converted 1 custom_function
```

- Convert all custom functions in the repository production and output to the repository local.

```
phenv customfunctions_to_py3 production local
```

Example output:

```
Converted custom_function "my_example_custom_function_01"
Converted custom_function "my_example_custom_function_02"

Successfully converted 2 custom_functions
```

playbooks_to_py3 options and examples

These are the options available when using the playbooks_to_py3 tool.

Argument	Required?	Description
repo/ repo2/playbook_name ...	Yes	A list of repositories or custom_functions to convert to Python 3. If repositories are listed, the tool will convert every playbook in the given repositories. Only provide the repository's name, not the full path to the repository.
repo	Yes	Output repository for converted playbooks.
-h, --help	No	Show the help message and exit the tool.
-d, --dry_run	No	Use this option to see which playbooks would be converted with the tool, without doing the conversion.
-s <SUFFIX>, --suffix <SUFFIX>	No	Specify a suffix to be added to the names of converted playbooks as part of the conversion process. If no suffix is specified the default is "_py3".
-v {0,1,2,3}, --verbosity {0,1,2,3}	No	Verbosity level: * 0 = minimal output * 1 = normal output * 2 = verbose output * 3 = very verbose output If you set --verbosity 3 you may see debug messages like this: Unsupported dict value of type: <class '_ast.NameConstant'>. Ignoring children and returning type./> These messages can be safely ignored. They are only informational messages from the code parser, and do not indicate a problem with the playbook.

Command examples:

- Convert the playbook example_find_attacks in repository production to Python3 and save it to the repository local. This will make a copy in the repository local called "example_find_attacks_py3".

```
phenv playbooks_to_py3 production/example_find_attacks local
```

If your playbook's name includes spaces, you need to enclose the name of the playbook in quotation marks on the command line.

```
phenv playbooks_to_py3 production/"example find attacks" local
```

Example output:

```
Converted playbook "example_find_attacks"

Successfully converted 1 playbook
```

- Convert multiple playbooks in the repository "qatest" to Python 3 and save them to the repository local:
  `phenv playbooks_to_py3 qatest/2_0_pb qatest/100_geolocates  local`
  Example output:

```
Converted playbook "2_0_pb"
Converted playbook "100_geolocates"

Successfully converted 2 playbooks
```

- Use a custom suffix "_my_suffix" to create a Python 3 copy of playbook called audit_test_my_suffix:

```
phenv playbooks_to_py3 qatest/audit_test local -s _my_suffix
```

Example output:

```
Converted playbook "audit_test"

Successfully converted 1 playbook
```

Troubleshooting for converting Splunk Phantom playbooks and custom functions from Python 2 to Python 3

If you run into issues when converting a playbook or custom function from Python 2 to Python 3, refer to the following tips:

- Resave the playbook.
  The conversion relies on how the playbook editor internally represents the playbook in JSON. When the playbook editor saves a playbook, it will update the format.
- Write code in the playbook block.
  If your new custom function validation fails, it might be validated as a legacy custom function if there is no code written in the playbook block. You might see errors such as: `ValueError: not enough values to unpack (expected 2, got 1)` or `AssertionError: Excepted two sections after splitting CF block`.

Automation API

Playbook automation API

The Splunk SOAR (On-premises) playbook automation API allows security operations teams to develop detailed automation strategies. Playbooks serve many purposes, ranging from automating small investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of the Splunk SOAR (On-premises) platform using playbooks.

act

The act API can be called from `on_start()` or the callback of any phantom.act() call. If multiple phantom.act() calls are called within the same function, they execute actions in parallel. If the action is executed on an asset that has primary approvers assigned or a reviewer specified, the action is not executed unless the primary approvers or reviewer approves the action.

The act API is not supported from within a custom function.

```
phantom.act(action, parameters=[], assets=None, tags=None,
            callback=None, reviewer=None, handle=None,
            start_time=None, name=None, asset_type=None,
            app=None)
```

Parameter	Required?	Description
action	Required	The name of the action that the user intends to run. Actions include block IP, list VM, or file reputation that are supported by the apps installed on the platform.
parameters	Optional	A list of dictionaries that contain the parameters expected by the action. The name of the keys are specific to the action being taken.
assets	Optional	A list of assets on which the action is run. If the user intends to take the action on a specific asset, it must be specified in this parameter. Assets are a list of asset IDs, as specified when an asset is configured. If the assets are configured with primary and secondary owners, the owners are required to approve an action before it can be run. If the asset is not specified, the action is run on all possible assets on which the action can be run. If multiple apps provide the same action for the same product, the system automatically uses the latest installed app. If new assets or apps are added to the Splunk SOAR platform, they might run actions that you hadn't intended to run. For example, if you begin your deployment with a simple network-based topology and configure a perimeter firewall that supports block IP, and then add an active directory (AD) server which has an associated app that also reads block IP, that action is run on both the firewall and AD server. Setting appropriate approvals on assets can help to minimize this risk.

tagsOptionalA list of asset tags that help specify certain assets to be used for executing the action. You can assign assets a tag when they are configured. For example, if the asset is tagged critical and the action is block IP, the action is run only on assets that are tagged as critical. If tags and assets are both specified, then the action is run only on assets tagged with the matching tag.callbackOptionalA specified callback function to be called upon completion of the action. Use the callback to evaluate the outcome of one action and then take more actions. Use the callback function to either serialize actions where you intend to run the actions one after the other, or where the subsequent action is dependent on the outcome or results of the first action.reviewerOptionalA username, email address, or group that receives an approval request to review the action before it is run. The user receives an approval request with all of the details of the action and its parameters. If Splunk SOAR (On-premises) is provided a comma- separated list or group, only one approval by any member of the list is required. SLA escalation settings affect how long the action is held for approval.handleOptionalA string object that, when specified, is passed on to the callback. Users can save any Python object that the user needs to access in the context of the callback from the action called. Handle is always saved with the action and passed to the callback. It is best to use handles to pass objects from action to callbacks instead of global variables.

The size of the handle object is limited to 4k. For objects bigger than 4k, use the save_data() and get_data() APIs instead.

start_timeOptionalThe time when the action is scheduled for execution. This value is a datetime object.

```
params=[]
params.append({'ip':'1.1.1.1'})

# schedule 'geolocate ip' 60 seconds from now
when = datetime.now()+timedelta(seconds=60)
phantom.act("geolocate ip",
    parameters=params,
    start_time=when)
```

nameOptionalA name the user can give to an instance of an action that is run.asset_typeOptionalUse the asset_type parameter to limit the action on assets of the specified type. This parameter can be a string or a list of strings.appOptionalThe specific app used to run the action. Specify the app as a Python dictionary: {"name":"some_app_name", "version":"x.x.x"}. "name" is case insensitive.

```
app_data={}
app_data['name']='MaxMind'
app_data['version']='1.1.0'
phantom.act('geolocate ip',
    parameters=[{ "ip" : "1.1.1.1" }],
    assets=["maxmind"],
    callback=geolocate_ip_cb,
    app=app_data)
```

or

```
phantom.act('geolocate ip',
    parameters=[{ "ip" : "1.1.1.1" }],
    assets=["maxmind"],
    callback=geolocate_ip_cb,
    app={'name':'MaxMind', 'version':'1.1.0' })
```

This sample playbook uses the phantom.act() API.

```
import phantom.rules as phantom
import json

def geolocate_ip_cb(action, success, container, results, handle):
    phantom.debug(results)
    if not success:
        phantom.debug('Action '+json.dumps(action)+' : FAILED')
        return
    return

def on_start(container):
    ips = set(phantom.collect(container, 'artifact:*.cef.sourceAddress'))
    parameters = []
    for ip in ips:
        parameters.append({ "ip" : ip })

    if parameters:
        phantom.act('geolocate ip', parameters=parameters, assets=["maxmind"], callback=geolocate_ip_cb)

    return

def on_finish(container, summary):
    return
```

callback

Callback functions are specified as parameters in phantom.act():

```
phantom.act('geolocate ip', parameters=parameters, assets=["maxmind"], callback=geolocate_ip_cb)
```

Callback functions are called when the phantom.act() action has completed, regardless if the action succeeded or failed.

The phantom.act API takes a callback function, which accepts the following keyword arguments:

- action
- success
- container
- results
- handle

The simplest no-op callback looks like this:

```
def do_nothing(action=None, success=None, container=None, results=None, handle=None):
    pass
```

Such a callback is functionally equivalent to not providing a callback parameter to phantom.act at all.

Parameter	Description

action	A JSON object that specifies the action name and the action run ID. The action run ID uniquely identifies that instance of action execution and can be used to retrieve the details of the action execution. The following example is an example of an action parameter from a callback:
	
	`{`
	`"action_run_id": 205,`
	`"action_name": "whois ip",`
	`"action" : "whois ip",`
	`"name": "my_whois_ip_action"`
	`}`
	The action field replaces the action_name field.

successEither true or false. An action is considered failed only if the action has failed on all assets and with all specified parameters. If any part of the action succeeds, it is not considered failed. Use the utility functions like `parse_errors()` or `parse_success()` to get a flat listing of all errors and success. These utility functions parse the results to give the user a different view of the overall results.containerThe container JSON object. Here is an example of container parameter from a callback:

```
{
    "sensitivity": "amber",
    "create_time": "2016-01-14 18:25:55.921199+00",
   "owner": "admin",
    "id": 7,
    "close_time": "",
    "severity": "medium",
    "label": "incident",
    "due_time": "2016-01-15 06:24:00+00",
    "version": "1",
    "current_rule_run_id": 1,
    "status": "open",
    "owner_name": "",
    "hash": "093d1d4d22cab1c5931bbfb1b16ce12c",
    "description": "this is my test incident",
    "tags": ["red-network"],
    "start_time": "2016-01-14 18:25:55.926468+00",
```

```
    "asset_name": "",
    "artifact_update_time": "2016-01-14 18:26:33.55643+00",
    "container_update_time": "2016-01-14 18:28:43.859814+00",
    "kill_chain": "",
    "name": "test",
    "ingest_app_id": "",
    "source_data_identifier":
        "48e4ab9c-2ec1-44a5-9d05-4e83bec05f87",
    "end_time": "",
    "artifact_count": 1
}
```

resultsA JSON object that has the full details and status of the complete action on all assets for each parameter. Here is an example where the geolocate ip action is run:

```
parameters = []
parameters.append({ "ip" : '1.1.1.1' })
phantom.act('geolocate ip', parameters=parameters,
    assets=["maxmind"], callback=geolocate_ip_cb,
    name='my_geolocate_action')
```

This simple action can result in various execution strategies and outcomes, depending on how the system is configured. In this simple form, one app supports the geolocate IP action and there is one Maxmind asset that is configured, which results in one IP being queried once on one asset. In a more complex example, if there are two apps, both of which support file reputation, then this one simple action results in a file hash queried on both of the assets.

The single action file reputation results in the hash being queried twice, once on each asset. The two queries still constitute one action, so the callback `file_reputation_cb` is called once when both the queries complete. The parameters in the example is a list of dictionaries and contains one IP. But, in one action call the user can specify many different IP addresses.

The `results` JSON object provides full visibility into the execution of the action on all matching assets using all matching apps for all specified parameters.

Here are the results of the geolocate IP action.

```
[
    {
        "asset_id": 63,
        "status": "success",
        "name": "geolocate_ip_1",
        "app": "MaxMind",
        "action_results": [
            {
                "status": "success",
                "data": [
                    {
                        "state_name": "Victoria",
                        "latitude": -37.7,
                        "country_iso_code": "AU",
                        "time_zone": "Australia/Melbourne",
                        "longitude": 145.1833,
                        "state_iso_code": "VIC",
                        "city_name": "Research",
                        "country_name": "Australia",
                        "continent_name": "Oceania",
                        "postal_code": "3095"
                    }
                ],
                "message": "City: Research, State: VIC,
```

```
                            Country: Australia",
                    "parameter": {
                        "ip": "1.1.1.1",
                        "context": {
                            "guid": "f42fd73f-...-8194aaa9bc11",
                            "artifact_id": 0,
                            "parent_action_run": []
                        }
                    },
                    "summary": {
                        "city": "Research",
                        "state": "VIC",
                        "country": "Australia"
                    }
                }
            ],
            "app_id": 83,
            "summary": {
                "total_objects": 1,
                "total_objects_successful": 1
            },
            "asset": "maxmind",
            "action": "geolocate ip",
            "message": "'geolocate_ip_1' on asset 'maxmind': 1
                        action succeeded. (1)For Parameter:
                        {\"ip\":\"1.1.1.1\"} Message: \"City:
                        Research, State: VIC, Country: Australia\"",
            "app_run_id": 51,
            "action_run_id": 11
        }
    ]
```

handleA string object that was specified in the action phantom.act() call for passing data between action and callbacks.

completed

The completed API checks if all of the provided runnables have finished running. Runnables are defined as actions, synchronous child playbooks, and custom functions. A runnable is finished running if its status is either succeeded or failed. Succeeded or failed implies that the action is done. If any combination of the action names, playbook names, or custom function names are not completed, then the function returns False. Use the completed API in the join function where certain blocks are run in parallel but the next block has to be called only when all the joining blocks have completed executing.

The completed API is not supported from within a custom function.

```
phantom.completed(action_names=None, playbook_names=None, custom_function_names=None, trace=False)
```

Parameter	Required?	Description
action_names	Optional	A list of names given to an action through the phantom.act() API in the parameter `name`.
playbook_names	Optional	A list of names given to a playbook execution using phantom.playbook() API in the parameter `name`.
custom_function_names	Optional	A list of names given to a custom function using the phantom.custom_function API in the parameter `name`.

This sample uses the completed API.

```
def join_add_tag_1(
    action=None,
    success=None,
```

```
    container=None,
    results=None,
    handle=None,
    filtered_artifacts=None,
    filtered_results=None,
):
    # Continue if all of the dependent blocks have completed
    if phantom.completed(
        action_names=['whois_ip_1'],
        playbook_names=['playbook_send_precautionary_email_1'],
        custom_function=['get_subnet_1'],
    ):
        # call subsequent block "add_tag_1"
        add_tag_1(container=container, handle=handle)
    return
```

condition

The condition API implements the decision block in the visual playbook editor (VPE). It evaluates expressions and returns matching artifacts and actions results that evaluate as true. Each filter block you create in the VPE calls condition.

The condition API is not supported from within a custom function.

```
phantom.condition(container=None,
                  action_results=None,
                  conditions=[],
                  logical_operator='or',
                  scope='new',
                  filtered_artifacts=[],
                  filtered_results=[],
                  limit=100,
                  name=None,
                  trace=False,
                  case_sensitive=True,
                  auto=True)
```

Parameter	Required?	Description
container	Required	The container dictionary object that is passed into the calling function.
action_results	Optional	The action results passed into any callback function or a subset of action results that had been filtered from a condition call. When you pass action results, you can also pass in custom function results. In other words, action results can be both action results and custom function results.

conditions	Required	A list of one or more `and` or `or` expressions to be evaluated. Matching artifacts or matching action results are returned.
		
		The following example shows the expression format:
		
		`[ [ LHS, OPERATOR, RHS ], [ LHS, OPERATOR, RHS ], ..]`
		
		`OPERATOR` can be any one of the following characters:
		
		`==, !=, <, >, <=, >=, in, not in`
		
		LHS and RHS values can be a value, artifact datapath, action result datapath, custom function result datapath, or a custom list datapath.

logical_operator	Optional	Expresses the relationship between conditions. Valid logical operators are `and` or `or`. Defaults to `or`.

Parameter	Required?	Description
scope	Optional	See collect. Possible values include `new`, `all`, or an artifact ID.
filtered_artifacts	Optional	Filtered artifacts that were returned from a preceding condition block.
filtered_results	Optional	Filtered results that were returned from a preceding condition block.
limit	Optional	See collect.
name	Optional	Specify a unique name to save the filtered action results and filtered artifacts to retrieve using either the collect2() API or phantom.get_filtered_data() API.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
case_sensitive	Optional	Default is True. Set to False'for evaluating conditions in a case-insensitive manner.
auto	Optional	A Boolean value where the default is True. When this value is True, remove the database record associated with the filtered data once the playbook run has finished.

The condition API returns a list of artifact IDs and a list of action result objects. These are the artifacts, actions results, and custom function results that match the conditions expressed. If you don't specify a filter statement about action results, no filtered action results or custom function results are returned and the VPE UI doesn't show that as a selectable option in subsequent blocks.

When using the VPE, you can select to connect various UI blocks. Each of these blocks implements a function in the auto-generated Python code. These functions have various parameters like container, results, filtered_artifacts, and filtered_results. The expressions used for the conditions can be either a constant or a datapath to specify what you need to retrieve and operate on. These datapaths can point to either a field in the artifact, action_result, filtered-artifacts, filtered results, or a constant.

To learn more about the datapaths used in this API, see Understanding datapaths.

Generally, CEF fields that are passed into the condition API, if they have commas in the value, cause the value to be treated as a list. If the CEF field is `toEmail` or `fromEmail`, then commas do not trigger the list behavior. This occurs because commas frequently appear in the display name portion of an email address.

If two strings that can be converted to a numeric type are being compared with one of the following operators, they are converted to numeric types before the comparison occurs:

```
==, !=, <, >, <=, >=
```

Example of condition

Here is some sample code that uses phantom.condition.

```
def filter_1(
    action=None,
    success=None,
    container=None,
    results=None,
    handle=None,
    filtered_artifacts=None,
    filtered_results=None,
    custom_function=None,
):
```

```
    action_results = [
        {
            'name':
                'normalize_ip_1',
            'action_results': [{
                'data': [{
                    'ip': '3.3.3.3',
                }],
                'parameter': {
                    'ip': '3.3.3.3\n',
                },
            }, ],
        },
        {
            'name':
                'normalize_ip_1',
            'action_results': [{
                'data': [{
                    'ip': '2.2.2.2',
                }],
                'parameter': {
                    'ip': '2.2.2.2\n',
                },
            }, ],
        },
    ]

    conditions = [
        [
            'normalize_ip_1:action_result.data.*.ip',
            '==',
            '2.2.2.2',
        ],
        [
            'normalize_ip_1:action_result.data.*.ip',
            '==',
            '0.0.0.0',
        ],
    ]

    # Call phantom.condition
    matched_artifacts_1, matched_results_1 = phantom.condition(container=container,
action_results=action_results, conditions=conditions, logical_operator='or')

    # The value of matched_results_1 is
    assert matched_results_1 == [
        {
            'name':
                'normalize_ip_1',
            'action_results': [{
                'data': [{
                    'ip': '2.2.2.2',
                }],
                'parameter': {
                    'ip': '2.2.2.2\n',
                },
            }, ],
        },
    ]

    # The value of matched_artifacts_1 is
```

```
    assert matched_artifacts_1 == []

    # Call the callback
    # The other parameters come from inputs into the block
    if matched_artifacts_1 or matched_results_1:
        domain_reputation_1(
            action=action,
            success=success,
            container=container,
            results=results,
            handle=handle,
            custom_function=custom_function,
            filtered_artifacts=matched_artifacts_1,
            filtered_results=matched_results_1,
        )
```

custom_function

Use the custom_function API to call a custom function from a playbook. The following table describes the parameters used in this function.

The custom_function API is not supported from within a custom function.

```
def custom_function(custom_function=None, parameters=None, callback=None, name=None):
```

Parameter	Description

custom_function	The custom function identifier. The Visual Playbook Editor (VPE) generates this identifier for you if you select your custom function through the configuration panel. Otherwise, use the following format:
	
	`# The custom function identifier specification`
	`<repository_name>/drafts?/<custom_function_name>`
	
	`# Valid identifier: <repository_name>/<custom_function_name>`
	`local/make_upper`
	`community/combine_datapaths`
	
	`# Valid identifier: <repository_name>/drafts/<custom_function_name>`
	`local/drafts/first_ten`
	
	`# Invalid identifier: missing a repository`
	`my_custom_function`
	
	`# Valid identifier, but drafts will be interpreted as a repository name`
	`drafts/my_custom_function`

parameters	A list of dictionaries containing the inputs to pass to the custom function callback. The shape of the dictionaries that are in the parameters list depends on what custom function you are calling.
name	The name of the custom function block. This is autogenerated by the VPE, but you can specify your own name from the configuration panel for the block using **Advanced Settings > General Settings > Name**. If you are not using the VPE, be aware that the name must be unique amongst all of the names in your playbook. For example, you cannot use the same name as an action elsewhere in the playbook.

callback	A callable object with a certain function signature. It is typically a function or possibly any Python callable. Invoke the object that you provide as the callback parameter as follows:
	
	`callback(`
	`container=container,`
	`results=result,`
	`handle=handle,`
	`custom_function=custom_function,`

Parameter	Description
	success=success) Your callable object must be able to accept these keyword arguments.

Example custom_function results object

The results parameter passed to the callback looks like this:

```
[
    {
        'custom_function_name': 'to_upper',
        'custom_function_results': [
            {
                'data': {
                    'upper': 'HELLO',
                },
                'parameter': {
                    'value': 'hello',
                },
            },
        ],
        'custom_function_run_id': 14,
        'message': '',
        'name': 'to_upper_1',
        'status': 'success',
    },
]
```

The length of the list corresponding to the `custom_function_results` key is the same as the length of the parameters list that was passed to the custom_function API.

callback

Callback functions are specified as parameters in the custom_function API:

```
phantom.custom_function('local/to_upper', parameters=[{'value': 'hello world'}], callback=decision_1)
```

The custom_function API takes a callback function, which accepts the following keyword arguments: `custom_function`, `success`, `container`, `results`, and `handle`. Although Python allows callers to pass keyword arguments in any order, customized callback functions accept the keyword arguments in the same order as previously listed, since Python also allows keyword arguments to be passed by position.

The simplest callback looks like this:

```
def do_nothing(custom_function=None, success=None, container=None, results=None, handle=None):
    pass
```

This callback is equivalent to not providing a callback parameter to phantom.custom_function at all.

Callback functions are called when the phantom.custom_function() action has completed, irrespective of action success or failure.

Parameter	Description
custom_function	

Parameter	Description

	A JSON object that specifies the metadata about the custom function that triggered the callback. The `custom_function_run_id` value corresponds to the object in the database that contains the data for the custom function run. You can give this ID to the phantom.get_custom_function_results API in order to retrieve the custom function results synchronously. The `name` value is the same as the `name` value passed to the triggering call of the API custom_function. It uniquely identifies the block name of the calling custom function. The `custom_function_name` parameter corresponds to the name of the triggering custom function.
	
	Here is an example `custom_function` parameter from a custom function callback:
	
	`{`
	`'custom_function_run_id': 22,`
	`'custom_function_name': 'phtest_cf_to_upper',`
	`'name': 'to_upper_1',`
	`}`

success	Returns as either true or false. A custom function always has a status of success unless it raises an uncaught exception.

container	The container JSON object. Here is an example of a container parameter from a callback:
	
	`{`
	`"sensitivity": "amber",`
	`"create_time": "2016-01-14 18:25:55.921199+00",`
	`"owner": "admin",`
	`"id": 7,`
	`"close_time": "",`
	`"severity": "medium",`
	`"label": "incident",`
	`"due_time": "2016-01-15 06:24:00+00",`
	`"version": "1",`
	`"current_rule_run_id": 1,`
	`"status": "open",`
	`"owner_name": "",`
	`"hash": "093d1d4d22cab1c5931bbfb1b16ce12c",`
	`"description": "this is my test incident",`
	`"tags": ["red-network"],`
	`"start_time": "2016-01-14 18:25:55.926468+00",`
	`"asset_name": "",`
	`"artifact_update_time": "2016-01-14 18:26:33.55643+00",`
	`"container_update_time": "2016-01-14 18:28:43.859814+00",`
	`"kill_chain": "",`
	`"name": "test",`
	`"ingest_app_id": "",`
	`"source_data_identifier":`
	`"48e4ab9c-2ec1-44a5-9d05-4e83bec05f87",`
	`"end_time": "",`
	`"artifact_count": 1`
	`}`

results	A JSON object that contains all of the custom function results produced by the triggering call to phantom.custom_function. Here is an example where the `phtest_to_upper` action is run:
	
	`[`
	`{`
	`"custom_function_name": "phtest_cf_to_upper",`
	`"custom_function_results": [`
	`{`
	`"data": {`
	`"upper": "hello world",`
	`},`

Parameter	Description
	"parameter": { "value": "hello world", }, },], "custom_function_run_id":22, "message": "", "name": "to_upper_1", "status": "success", },]
handle	A string object that is specified in the action phantom.custom_function() call for passing data between custom functions and callbacks.
status	The status of the custom function that was run. Status returns as either success or fail.

debug

When logging is enabled, the debug API lets the author debug as the playbook is being developed and tested. This is similar to a print() statement. The parameter for the call is a string type object and the contents are shown in the debug console in cyan text so that you can distinguish your text from the system text.

The debug API is supported from within a custom function.

```
phantom.debug(message)
```

The following example shows the debug API:

```
def on_start(container):
    phantom.debug('in on_start() of playbook')
    return
```

The response looks something like this:

```
2016-02-13T01:32:52.583000+00:00: calling on_start(): on incident 'test', id: 107.
2016-02-13T01:32:52.608695+00:00: in on_start() of playbook
```

The debug and error APIs encode arguments to UTF-8 before printing them. If debug is passed a Python list or a dictionary at any level of nesting, it decodes any unicode strings within that mutable object. This means that calling debug or error can mutate the argument passed to those functions. To work around this behavior, do a deep copy of the object that you want to debug and pass the copy to phantom.debug as shown in the following example:

```
from copy import deepcopy

names = [u'Jos<', u'Mar>a', u'Rosa']
names_copy = deepcopy(names)

# Debug print names_copy, thus preserving names
phantom.debug(names_copy)

# Due to the bug, names_copy has been mutated
assert names != names_copy

assert names == [u'Jos<', u'Mar>a', u'Rosa']
```

decision

Decision blocks in the Visual Playbook Editor generate calls to the decision API. The decision API returns a Boolean value to indicate decision success or failure.

The decision API is a mechanism of control flow so it can't be called from within a custom function.

```
phantom.decision(container=None,
                 action_results=None,
                 conditions=[],
                 logical_operator='or',
                 scope='new',
                 filtered_artifacts=[],
                 filtered_results=[],
                 limit=100,
                 name=None,
                 trace=False,
                 case_sensitive=True,
                 auto=True)
```

Parameter	Required?	Description
container	Required	The container dictionary object that is passed into the calling function.
action_results	Optional	The action results passed into any callback function or a subset of action results that were filtered from a phantom.condition() call. When you pass in action results, you can also pass in custom function results. Action results can be both action results and custom function results.

conditions	Required	A list of one or more `and` or `or` expressions to be evaluated. Matching artifacts or matching action results are returned.
		
		The following example shows the expression format:
		
		`[ [ LHS, OPERATOR, RHS ], [ LHS, OPERATOR, RHS ], ..]`
		
		`OPERATOR` can be any one of the following characters:
		
		`==, !=, <, >, <=, >=, in, not in`
		
		LHS and RHS values can be a value, artifact datapath, action result datapath, custom function result datapath, or a custom list datapath.

logical_operator	Optional	Expresses the relationship between conditions. Valid logical operators are `and` or `or`. Defaults to `or`. If the logical operator is `and` then each expression passed to the condition must be true on the same result, if the expression relates to a result, for decision to return true. Potential result types are artifacts, action results, or custom function results.
scope	Optional	See the collect API documentation. Possible values include `new`, `all`, or an artifact ID.
filtered_artifacts	Optional	Filtered artifacts that were returned from a preceding phantom.condition() block.
filtered_results	Optional	Filtered results that were returned from a preceding phantom.condition() block.
limit	Optional	This enforces the maximum number of artifacts that can be retrieved in this call. The default is 100.
name	Optional	Specify a unique name to save the filtered action results and filtered artifacts which can be retrieved using either the collect2() API or the phantom.get_filtered_data() API.
trace	Optional	

Parameter	Required?	Description
		Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
case_sensitive	Optional	Default is True. Set to False for evaluating conditions in a case-insensitive manner.
auto	Optional	A Boolean value where the default is True. When this value is True, remove the database record associated with the filtered data once the playbook run has finished.

Generally, CEF fields that are passed into the decision API, if they have commas in the value, cause the value to be treated as a list. One specific exception is if the CEF field is `toEmail` or `fromEmail` where commas do not trigger the list behavior because commas frequently appear in the display name portion of an email address.

When passing Boolean values to a decision block in the Visual Playbook Editor, true and false with lowercase letters, are interpreted as strings. while True and False with capital letters, are interpreted as Boolean values. These capitalization conventions match Python, and True and False are built in, but true and false are not.

If two strings that can be converted to a numeric type are being compared with one of the following operators, they are converted to numeric types before the comparison occurs:

```
==, !=, <, >, <=, >=
```

Example of decision

Here is some sample code that uses phantom.decision.

```
def decision_2(
    action=None,
    success=None,
    container=None,
    results=None,
    handle=None,
    filtered_artifacts=None,
    filtered_results=None,
    custom_function=None,
):

    action_results = [
        {
            'name':
                'normalize_ip_1',
            'action_results': [{
                'data': [{
                    'ip': '3.3.3.3',
                }],
                'parameter': {
                    'ip': '3.3.3.3\n',
                },
            }, ],
        },
        {
            'name':
                'normalize_ip_1',
            'action_results': [{
                'data': [{
                    'ip': '2.2.2.2',
                }],
                'parameter': {
```

```
                    'ip': '2.2.2.2\n',
                },
            }, ],
        },
    ]
    conditions = [
        [
            'normalize_ip_1:action_result.data.*.ip',
            '==',
            '2.2.2.2',
        ],
        [
            'normalize_ip_1:action_result.data.*.ip',
            '==',
            '0.0.0.0',
        ],
    ]

    # Call to the phantom.decision API
    # With logical_operator set to 'or'
    matched = phantom.decision(container=container, action_results=action_results, conditions=conditions,
logical_operator='or')

    # Return value is True
    assert matched is True

    # Call to the phantom.decision API
    # With logical_operator set to 'and'
    phantom.decision(container=container, action_results=action_results, conditions=conditions,
logical_operator='and')

    # Return value is False
    assert matched is False


    # Call to the phantom.decision API
    # with literal conditions
    phantom.decision(container=container, action_results=action_results, conditions=[[4, '==', 4], [True,
'!=', False], logical_operator='and')

    # Return value is True
    assert matched is True

    # Call the callback function
    if matched:
       send_email_1(
            action=action,
            success=success,
            container=container,
            results=results,
            handle=handle,
            custom_function=custom_function,
        )
```

discontinue

The discontinueAPI allows the users to stop executing active playbooks when a container is being processed against Active playbooks.

The discontinue API is not supported from within a custom function.

```
phantom.discontinue()
```
Example:

```
def on_start(container):
    phantom.discontinue()
```

error

The error API lets the author debug or print log messages as the playbook is run with logging disabled. This is similar to a print() statement. The parameter for the call is a string type object and the contents are shown in the playbook debug console in red text so that you can distinguish your text from the system text.

The error API is supported from within a custom function.

```
phantom.error(message)
```
The following example shows the error API:

```
def on_start(container):
    phantom.error('in on_start() of playbook')
    return
```
The response looks like something like this:

```
2016-02-13T01:32:52.583000+00:00: calling on_start(): on incident 'test', id: 107.
2016-02-13T01:32:52.608695+00:00: in on_start() of playbook
```

The error and debug APIs encode arguments to UTF-8 before printing them. If phantom.debug is passed a Python list or a dictionary at any level of nesting, it decodes any unicode strings within that mutable object. This means that calling debug or error can mutate the argument passed to those functions. To work around this behavior, do a deep copy of the object that you want like to debug and pass the copy to debug as shown in the following example:

```
from copy import deepcopy

names = [u'Jos<', u'Mar>a', u'Rosa']
names_copy = deepcopy(names)

# Debug print names_copy, thus preserving names
phantom.debug(names_copy)

# Due to the bug, names_copy has been mutated
assert names != names_copy

assert names == [u'Jos<', u'Mar>a', u'Rosa']
```

format

The format API formats text with values that are extracted using datapaths for other complex objects such as artifacts or action results.

The format API is supported from within a custom function.

```
 phantom.format(container=None,
                template=None,
                parameters=None,
```

```
                scope='new',
                name=None,
                trace=False,
                separator=None,
                drop_none=False):
```

Parameter	Required?	Description
container	Required	The container object passed into the action callback or `on_start`.
template	Required	The format string where positional arguments are substituted with values. The arguments are expressed and passed as a list of datapaths in the `parameters` argument. If the datapath returns a list of items, the positional argument is replaced by a comma-separated value of the items. The format string uses positional arguments that are the same as Python string formatting.
parameters	Required	A list of datapaths with a corresponding datapath for each positional format argument used in the template string.
scope	Optional	See collect for more information. The default value for scope is `new`, but the values can be either `all` or `new`.
name	Optional	The name used to save the resulting formatted data. Use this name to retrieve this parameter through the get_format_data() API. If this parameter is not specified, the data is not saved.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
separator	Optional	If a datapath response contains a list of strings or numbers, but not Python objects, the default output separator is `', '`. You can specify an alternate separator using this parameter.
drop_none	Optional	The default value is `False`. By default `None` values are included in the resulting string, but the user can filter `None` type values in the resulting string through this parameter.

Example request

This sample uses the phantom.format API.

```
def on_start(container):

    template = "Host '{0}' transferred in '{1}' bytes"

    datapaths = ['artifact:*.cef.sourceAddress','artifact:*.cef.bytesIn']

    formatted_data = phantom.format(container=container,
                                    template=template,
                                    parameters=datapaths,
                                    name='formatted_1')

    phantom.debug("Formatted data is: {}".format(formatted_data))

    # use the same name that was provided for key in phantom.format()
    formatted_data = phantom.get_format_data(name="format_1")

    phantom.debug("Retrieved formatted data is: {}".format(formatted_data))

    return
```

Example response

The following is an example of what the format API returns.

```
Starting playbook 'format_test' on 'incident' id '13' with playbook run id '7'.
calling on_start() on incident 'test_incident'(id: 13).
phantom.collect2(): called for datapath['artifact:*.cef.sourceAddress']
phantom.collect2(): called for datapath['artifact:*.cef.bytesIn']
save_run_data() called
Formatted data is: Host '1.1.1.1' transferred in '999' bytes
get_run_data() called
Retrieved formatted data is: Host '1.1.1.1' transferred in '999' bytes
No actions were executed
calling on_finish()


Playbook 'format_test' (playbook id: 6) executed (playbook run id: 7) on
        incident 'test_incident'(container id: 13).
           Playbook execution status is 'success'
        Total actions executed: 0

{"message":"No actions were executed","playbook_run_id":7,"result":[],
        "status":"success"}
```

In the previous example, if there are multiple artifacts `artifact:*.cef.sourceAddress` refers to a list of IPs, and the output looks like the following:

Retrieved formatted data is: Host '1.1.1.1', '8.8.8.8', '8.8.4.4' transferred in '999', '888', '777' bytes

For each pair of sourceAddress and bytesIn to have their own line in this output, wrap sections of the format text in %%, as shown in this sample.

```
def on_start(container):

    template = """\
%%
Host '{0}' transferred in '{1}' bytes"
%%"""

    datapaths = ['artifact:*.cef.sourceAddress','artifact:*.cef.bytesIn']

    formatted_data = phantom.format(container=container,
                                    template=template,
                                    parameters=datapaths,
                                    name='formatted_1')

    phantom.debug("Formatted data is: {}".format(formatted_data))

    # use the same name that was provided for key in phantom.format()
    formatted_data = phantom.get_format_data(name="format_1")

    phantom.debug("Retrieved formatted data is: {}".format(formatted_data))

    return
```

This creates the following string:

```
Retrieved formatted data is: Host '1.1.1.1' transferred in '999' bytes
Host '8.8.8.8' transferred in '888' bytes
Host '8.8.4.4' transferred in '777' bytes
```

When creating this string, each section is saved into a list. When using the VPE, you can get this list for the format block, and then each item in this list will be passed a parameter to the action as shown in the following example.

```
def on_start(container):

    template = """\
%%
Host '{0}' transferred in '{1}' bytes"
%%"""

    datapaths = ['artifact:*.cef.sourceAddress','artifact:*.cef.bytesIn']

    # formatted_data will always be a string
    formatted_data = phantom.format(container=container,
                                    template=template,
                                    parameters=datapaths,
                                    name='formatted_1')

    # Add __as_list to the end of the name to retrieve the list instead of the string
    formatted_data_list = phantom.get_format_data(name="format_1__as_list")
    return
```

playbook

The playbook API enables users to call another playbook from within the current playbook. If there are two or more playbooks with the same name from different repositories, the call fails. As such, use the format "repo_name/playbook_name" to be specific. The playbook API returns the playbook_run_id that can be used to query corresponding playbook execution details and report.

The playbook API is not supported from within a custom function.

```
phantom.playbook(playbook=None,
                 container=None,
                 handle=None,
                 show_debug=False,
                 callback=None,
                 inherit_scope=True,
                 inputs=None,
                 name=None)
```

Parameter	Required?	Description
playbook	Required	The playbook name to run. Use the format "repo_name/playbook_name".
container	Required	The container JSON object that needs to be passed to run the playbook on. This is the same container JSON object that you get in `on_start()` or any other callback function.
handle	Optional	An object that you can pass to the API that is passed back to the callback when the playbook finishes execution.
show_debug	Optional	The default for this parameter is False, but if you set it to True, the debug messages of the launched playbook is shown in the debug window when you debug the caller playbook.
callback	Optional	If this parameter is specified, the playbook is launched in a synchronous fashion. When the child playbook finishes, the specified callback function is called with playbook execution results. When child playbooks are launched synchronously, the parent playbook is not considered completed until the called child playbook has finished executing. If this parameter is specified, you must also specify the `name` parameter.
inherit_scope	Optional	Default is True. This parameter implies that the child playbook inherits the scope settings from the parent when called. If set to false, the child playbook runs with the default playbook scope.
inputs	Optional	If specified, this value must be a JSON-serializable dictionary of one or more key/value pairs, passed as an input for the child playbook that is called. Find the input specifications in the `input_spec` field, either in the user interface or in the REST API. Be sure both of the following are true:

Parameter	Required?	Description
		• The child playbook that is called must be an input playbook, not an Automation playbook. • The inputs you provide must be valid for the child playbook.
name	Required	An optional parameter unless the callback parameter is specified. This parameter identifies the execution instance of the called playbook. If the code for calling the child playbook is auto-generated, the name of the function is the recommended value for this parameter.

This sample playbook shows calling a playbook from a playbook.

```
"""
This sample playbook shows calling a playbook from a playbook
"""
import json
import phantom.rules as phantom

def playbook_file_analysis_1(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None, custom_function=None, **kwargs):
    phantom.debug("playbook_file_analysis_1() called")

    inputs = {
        "file_hash": "09ca7e4eaa6e8ae9c7d261167129184883644d07dfba7cbfbc4c8a2e08360d5b",
        "file_name": "hello.txt",
    }

    # call playbook "local/file_analysis", returns the playbook_run_id
    playbook_run_id = phantom.playbook("local/file_analysis", container=container, inputs=inputs)

    return
```

prompt

Using phantom.prompt() results in a message sent to the specified approvers.

- Approvers can be users or roles.
- A notification is sent to the notification bell icon in the upper right corner of the screen for the specified approvers.
- If Splunk SOAR (On-premises) has been configured with an SMTP asset, and the approvers have valid email addresses in their account settings, the approvers are sent an email.

  An email notification sent using phantom.prompt or phantom.prompt2 cannot be disabled by Splunk SOAR users by disabling notifications in their account settings.

Pending notifications can be accessed by clicking the bell icon in the top right corner of the UI. Delegation for approvals is possible. Either the approver or a delegate can complete the task. When the task is completed, the prompt() callback is called with the final response included in the action results.

The structure of the callback function and all the parameters is consistent with an action callback. Using prompt only allows the user to complete a task. The playbook can contain a callback function and use the prompt response, found in the result object in the callback, to change playbook behavior.

The prompt API is not supported from within a custom function.

```
 phantom.prompt(user=None,
                message='',
                respond_in_mins=30,
                callback=None,
```

```
                name=None,
                options=None,
                parameters=None,
                container=None,
                scope='new',
                trace=False,
                separator=None,
                drop_none=False)
```

Parameter	Required?	Description
container	Required	The object that is associated with the current playbook execution. This object is available to all action callbacks and other playbook execution functions.
user	Required	The recipient in the form of a user email address, username, or a role. Must be a valid user or role in Splunk SOAR (On-premises).
message	Required	The message content to send.
respond_in_mins	Optional	The time the user is given to respond. Default is 30 minutes. If the user does not respond in the specified time, the prompt fails and a failed status is sent to the callback.
callback	Optional	This parameter the same prototype as action callbacks. Status indicates success when the user has responded to the action and is failure only when the user does not respond in the specified time. The results JSON has the same format as any action results. Handle is not used and is an empty object.
name	Optional	The name of the action.
options	Optional	A JSON dictionary. Allows the user response to display with programmed choices.
parameters	Optional	A list of datapaths whose values are used to format the message. Recognized datapaths are used to retrieve data, and the data is used to populate the curly brackets in the message. The first parameter replaces {0}, the second replaces {1}, and so on.
scope	Optional	The scope can either be `new` or `all`. Default value is `new`. See collect for more information.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
separator	Optional	Specify an alternate separator using this parameter. If a datapath response contains a list, the default output separator is `', '`.
drop_none	Optional	By default, the `None` values are included in the resulting string.

This sample playbook shows calling a phantom.prompt() from a playbook.

```
"""
This sample playbook shows calling a phantom.prompt() from a playbook
"""
import phantom.rules as phantom
import json


def on_start(container):
    phantom.prompt(container=container,
                   user="user@company.com",
                   message="proceed with blocking these ips on FW?",
                   respond_in_mins=10,
                   callback=prompt_cb,
                   options={ 'type': 'list', 'choices': ['yes', 'no', 'maybe'] },
                   name="prompt_to_block_ips")
    return

def prompt_cb(action, success, container, results, handle):
    phantom.debug(results)
```

```
    return


def on_finish(container, summary):
    return
```

The following shows the output of the playbook:

```
Fri Apr 29 2016 19:38:25 GMT-0700 (PDT): Starting playbook 'manual_action' testing on 'incident' id:
'215'...
Fri Apr 29 2016 19:38:25 GMT-0700 (PDT): calling on_start(): on incident 'test', id: 215.
Fri Apr 29 2016 19:38:25 GMT-0700 (PDT): phantom.act(): Warning: For action 'prompt' no assets were
specified. The action shall execute on all assets the app (supporting the action) can be executed on
Fri Apr 29 2016 19:38:25 GMT-0700 (PDT): phantom.act(): action 'prompt' shall be executed with parameters:
'[{"to": "user@company.com", "message": "please make sure xyz is ok ...", "mins_to_act": 10}]', assets: '',
callback function: 'prompt_cb', with no action approver, no delay to execute the action, no user provided
name for the action, no tags, no asset type
Fri Apr 29 2016 19:38:25 GMT-0700 (PDT): Request sent  for action'automated action 'prompt' of
'manual_action' playbook'
Fri Apr 29 2016 19:38:50 GMT-0700 (PDT): Manual action was completed by the user. User message: yes I am
OK..
Fri Apr 29 2016 19:38:50 GMT-0700 (PDT): calling action callback function: prompt_cb
Fri Apr 29 2016 19:38:50 GMT-0700 (PDT):

[
    {
        "asset_id": 0,
        "status": "success",
        "name": "prompt_to_block_ips",
        "app": "",
        "action_results": [
            {
                "status": "success",
                "data": [
                    {
                        "response": "maybe"
                    }
                ],
                "message": "proceed with blocking these ips on FW?",
                "parameter": {
                    "message": "proceed with blocking these ips on FW?"
                },
                "summary": {
                    "response": "maybe"
                }
            }
        ],
        "app_id": 0,
        "app_run_id": 0,
        "asset": "",
        "action": "prompt",
        "message": "1 action succeeded",
        "summary": {},
        "action_run_id": 57
    }
]

Fri Apr 29 2016 19:38:50 GMT-0700 (PDT): successfully called action callback 'prompt_cb()' in rule:
manual_action(id:182)
Fri Apr 29 2016 19:38:50 GMT-0700 (PDT): calling on_finish()
Fri Apr 29 2016 19:38:50 GMT-0700 (PDT):
Playbook 'manual_action (id: 182)' executed (playbook_run_id: 195) on incident 'test'(id: 215).
```

```
Playbook execution status is:'success'
        No actions were executed for this playbook and 'incident'
Fri Apr 29 2016 19:38:50 GMT-0700 (PDT): *** Playbook execution has completed with status: SUCCESS ***
Fri Apr 29 2016 19:38:51 GMT-0700 (PDT): Playbook execution report:
{"message":"","playbook_run_id":195,"result":[{"action":"prompt","app_runs":null,"close_time":"2016-04
-30T02:38:50.844839+00:00","create_time":"2016-04-30T02:38:25.731+00:00","id":156,"message":"yes I am OK..
","name":"automated action 'prompt' of 'manual_action'
playbook","status":"success","type":"manual"}],"status":"success"}
```

Option Parameter Examples

```
Type: list
Shows the items in choices as the availabe responses.
{ 'type': 'list', 'choices': ['High', 'Medium', 'Low'] }

Type: range
Shows an input that requires a response within the given range of integers, i.e. 1-10.
{ 'type': 'range', 'min': 1, 'max': 100 }

Type: message
Shows a text area input in which a free form response can be entered.
{ 'type': 'message'}
```

prompt2

The prompt2 API is similar to the prompt API, but with prompt2 you can create a prompt with multiple user input fields. In prompt2, the `options` parameter is replaced by the `response_types` parameter. The other parameters are the same as in the prompt API. See prompt.

The prompt2 API is not supported from within a custom function.

```
phantom.prompt2(user=None,
                message='',
                respond_in_mins=30,
                response_types=None,
                callback=None,
                name=None,
                parameters=None,
                container=None,
                scope='new',
                trace=False,
                separator=None,
                drop_none=False)
```

Parameter	Required?	Description
container	Required	The container object associated with the current playbook execution. This object is available to all action callbacks and other playbook execution functions.
user	Required	The recipient in the form of a user email address, username, or a role. Must be a valid user or role in Splunk SOAR (On-premises).
message	Required	The message content to send.
response_types	Required	The list of JSON dictionaries describing each input field in the prompt.
respond_in_mins	Optional	The time the user is given to respond. The default is 30 minutes. If the user does not respond in the specified time, the prompt fails and a failed status is sent to the callback.
callback	Optional	This parameter has the same prototype as action callbacks. Status indicates success when the user has responded to the action and is failure only when the user does not respond in the specified time. The results

Parameter	Required?	Description
		JSON has the same format as any action results. Handle is not used and is an empty object.
name	Optional	The name of the prompt.
parameters	Optional	A list of datapaths whose values are used to format the message. Recognized datapaths are used to retrieve data, and the data is used to populate the curly brackets in the message. The first parameter replaces {0}, the second replaces {1}, and so on.
scope	Optional	Can either be `new` or `all`. Default value is `new`. See collect.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
separator	Optional	Specify an alternate separator using this parameter. If a datapath response contains a list, the default output separator is `', '`.
drop_none	Optional	By default, the `None` values are included in the resulting string.

Example of prompt2 response types

Response types are a list of JSON objects. Each object represents one input field in the prompt to be created. Each object needs to have a value with the key `prompt`, and has an optional key, `options`. `prompt` is a message for that input field. `options` is a dictionary with the same structure as a dictionary for the `options` in the prompt API. No options being specified results in a normal message type prompt.

```
[
  {'prompt': 'Select a number in this range', 'options': {'type': 'range', 'min': 1, 'max': 50}},
  {'prompt': 'Describe the event'}
]
```

The `message` parameter is still used in the prompt2 API. The message is displayed at the top of the created prompt, before the input fields.

render_template

The render_template API accepts a Django 1.11 template and fills the variable fields with contextual information from a provided dictionary. Common uses of the template are for user prompts or case management updates. Additional information about Django 1.11 templates can be found by searching on the Django Project home page.

The render_template API is supported from within a custom function.

```
phantom.render_template(template, context)
```

Parameter	Required?	Description
template	Required	The Django 1.11 template.
context	Required	Dictionary of values used to populate variable fields in the Django template.

This sample demonstrates the addition and population of a template in a playbook.

```
phantom.render_template(
        "<html>
            <head>
                <title>Report for {{ report_name }}</title>
            </head>
            <body>Hi {{ subject }}, here are a list of IPs you should look at! <ul>{% for ip in ip_list %}
                <li>{{ ip }}</li>
```

```
                {% endfor %} </ul>
            </body>
        </html>",
        {
        'report_name': 'Task for {}: {}'.format(container['id'],
        container['name']),
        'subject': container['owner_name'],
        'ip_list': ips_affected
        }
    )
```

task

The task API is a specialization of a manual action to ask a user or a role to perform work in the course of a response workflow or playbook.

The task API is not supported from within a custom function.

```
phantom.task(user=None, message=None, respond_in_mins=0, callback=None, name=None)
```

Parameter	Required?	Description
user	Required	The person or a role to whom the task is assigned.
message	Required	The text that has the information or details of the task.
respond_in_mins	Required	The time given to the user to perform the task, after which the task fails and the status is expressed in the callback if it was specified.
callback	Optional	A callback function to be called when the task completes.
name	Required	A unique name to distinguish this action from other actions

Container automation API

The Splunk SOAR (On-premises) Automation API allows security operations teams to develop detailed and precise automation strategies. Playbooks can serve many purposes, ranging from automating minimal investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of data management automation using playbooks.

add_artifact

Add a new artifact to the specified container. After creating an artifact, this call returns a tuple of a success flag (Boolean), any response message (string), and the artifact ID (integer).

The add_artifact API is supported from within a custom function.

```
phantom.add_artifact(container=None, raw_data=None, cef_data=None, label=None, name=None,
    severity=None, identifier=None, artifact_type=None,
    field_mapping=None, trace=False, run_automation=False)
```

Parameter	Required?	Description
container	Optional	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
raw_data	Required	The raw artifact data is stored as-is and is accessible from the artifacts. You can also use an empty dictionary.
cef_data	Required	The Common Event Format (CEF) representation of the original data. You can also use an empty dictionary.

Parameter	Required?	Description
label	Required	The label expressing the class of artifact. For example, event or netflow.
name	Required	The name of the artifact.
severity	Required	Supported values are `high`, `medium`, `low`, or the name of any active severity on the system.
identifier	Required	The source data identifier for the artifact. Generally this is a unique ID from a data source. A value of `None` generates a GUID for the source data identifier.
artifact_type	Required	The type of the artifact, such as `host` or `network`.
field_mapping	Optional	If CEF field names are not specific enough, use this parameter to specify the contains type for the field. For example, if you add a playbook with an artifact that has a CEF field named `user_hash`, this parameter can help convey to the platform that `user_hash` is a hash type of field that can then be used to show contextual actions.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
run_automation	Optional	Default is `False`. If set to `True`, the parameter causes active playbooks to run when new artifacts are added to the container.

This sample uses the phantom.add_artifact() API.

```
import phantom.rules as phantom
import json
import uuid

def on_start(container):

    artifacts = phantom.collect(container, 'artifacts:*', scope='all')
    phantom.debug(artifacts)

    raw = {}
    cef = {}
    cef['sourceAddress'] = '1.1.1.1'

    success, message, artifact_id = phantom.add_artifact(
        container=container, raw_data=raw, cef_data=cef, label='netflow',
        name='test_event', severity='high',
        identifier=None,
        artifact_type='network')

    phantom.debug('artifact added as id:'+str(artifact_id))

    artifacts = phantom.collect(container, 'artifacts:*', scope='all')
    phantom.debug(artifacts)

    # optionally user can specify a type for custom CEF fields.
    cef['foo'] = 'c8e5728b05c3ac46212c33535b65f183'
    field_mapping = {}
    field_mapping['foo'] = ['md5']

    success, message, artifact_id= phantom.add_artifact(
        container=container, raw_data=raw, cef_data=cef, label='netflow',
        name='test_event', severity='high',
        identifier=None,
        artifact_type='network',
        field_mapping=field_mapping)

    phantom.debug('artifact added as id:'+str(artifact_id))
```

```
    return


def on_finish(container, summary):

    return
```

The output of the playbook in the debugger shows the following results:

```
Thu May 05 2016 14:03:42 GMT-0700 (PDT): Starting playbook '2498' testing on 'incident' id: '79'...
Thu May 05 2016 14:03:42 GMT-0700 (PDT): calling on_start(): on incident 'test_incident', id: 79.
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.collect() called with datapath: artifacts:*, limit = 100
and none_if_first=False
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.collect(): will be limited to return 100 rows by default.
To override, please provide limit parameter. 0 implies no limit
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.collect():  called with datapath 'artifacts:*', scope='all'
and limit=100. Found 0 TOTAL artifacts
Thu May 05 2016 14:03:42 GMT-0700 (PDT):
[]
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.add_artifact() called
Thu May 05 2016 14:03:42 GMT-0700 (PDT): artifact added as id:122
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.collect() called with datapath: artifacts:*, limit = 100
and none_if_first=False
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.collect(): will be limited to return 100 rows by default.
To override, please provide limit parameter. 0 implies no limit
Thu May 05 2016 14:03:42 GMT-0700 (PDT): phantom.collect():  called with datapath 'artifacts:*', scope='all'
and limit=100. Found 1 TOTAL artifacts
Thu May 05 2016 14:03:43 GMT-0700 (PDT):
[
    {
        "create_time": 1462482222615,
        "id": 122,
        "severity": "high",
        "label": "netflow",
        "version": 1,
        "type": "network",
        "owner_id": 0,
        "cef": {
            "sourceAddress": "1.1.1.1"
        },
        "update_time": 1462482222615,
        "hash": "4cb608956567e4a95784382de5f81c1b",
        "description": "",
        "tags": [],
        "cef_types": {},
        "start_time": 1462482222615,
        "container_id": 79,
        "kill_chain": "",
        "playbook_run_id": 25,
        "data": {},
        "name": "test_event",
        "ingest_app_id": 0,
        "source_data_identifier": "0617e876-7ca1-407d-bab4-b5c3acf644f3",
        "end_time": 0
    }
]
Thu May 05 2016 14:03:43 GMT-0700 (PDT): phantom.add_artifact() called
Thu May 05 2016 14:03:43 GMT-0700 (PDT): artifact added as id:123
Thu May 05 2016 14:03:43 GMT-0700 (PDT): phantom.collect() called with datapath: artifacts:*, limit = 100
and none_if_first=False
Thu May 05 2016 14:03:43 GMT-0700 (PDT): phantom.collect(): will be limited to return 100 rows by default.
```

```
To override, please provide limit parameter. 0 implies no limit
Thu May 05 2016 14:03:43 GMT-0700 (PDT): phantom.collect():  called with datapath 'artifacts:*', scope='all'
and limit=100. Found 2 TOTAL artifacts
Thu May 05 2016 14:03:43 GMT-0700 (PDT):
[
    {
        "create_time": 1462482222615,
        "id": 122,
        "severity": "high",
        "label": "netflow",
        "version": 1,
        "type": "network",
        "owner_id": 0,
        "cef": {
            "sourceAddress": "1.1.1.1"
        },
        "update_time": 1462482222615,
        "hash": "4cb608956567e4a95784382de5f81c1b",
        "description": "",
        "tags": [],
        "cef_types": {},
        "start_time": 1462482222615,
        "container_id": 79,
        "kill_chain": "",
        "playbook_run_id": 25,
        "data": {},
        "name": "test_event",
        "ingest_app_id": 0,
        "source_data_identifier": "0617e876-7ca1-407d-bab4-b5c3acf644f3",
        "end_time": 0
    },
    {
        "create_time": 1462482223566,
        "id": 123,
        "severity": "high",
        "label": "netflow",
        "version": 1,
        "type": "network",
        "owner_id": 0,
        "cef": {
            "foo": "c8e5728b05c3ac46212c33535b65f183",
            "sourceAddress": "1.1.1.1"
        },
        "update_time": 1462482223566,
        "hash": "03134c15c2df2f7417f3ce360ce75d87",
        "description": "",
        "tags": [],
        "cef_types": {
            "foo": [
                "md5"
            ]
        },
        "start_time": 1462482223566,
        "container_id": 79,
        "kill_chain": "",
        "playbook_run_id": 25,
        "data": {},
        "name": "test_event",
        "ingest_app_id": 0,
        "source_data_identifier": "8dc4e4f8-0d81-4c64-8685-4742bd1d7bce",
        "end_time": 0
    }
```

```
]
Thu May 05 2016 14:03:43 GMT-0700 (PDT): No actions were executed
Thu May 05 2016 14:03:43 GMT-0700 (PDT): calling on_finish()
Thu May 05 2016 14:03:43 GMT-0700 (PDT):
Playbook '2498 (id: 58)' executed (playbook_run_id: 25) on incident 'test_incident'(id: 79).
Playbook execution status is:'success'
        No actions were executed for this playbook and 'incident'
Thu May 05 2016 14:03:43 GMT-0700 (PDT): {"message":"No actions were
executed","playbook_run_id":25,"result":[],"status":"success"}
```

When the playbook calls this API, the newly added artifact is not accessible to this instance of the playbook run through the phantom.collect() API id scope='new'. The previous sample uses scope='all' in the phantom.collect() call to get all the artifacts including the one that was added to the container.

add_note

Use the add_note API to add a note of the specified type to the container. On completion it returns a tuple of a success flag, any response messages, and the note ID.

```
phantom.add_note(container=None, note_type=None, note_format=markdown, trace=False)
```

The add_note API is supported from within a custom function.

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
note_type	Required	String	The type of note you want to create. Valid options are "general", "artifact", or "task". For "artifact" or "task" types you must supply the `artifact_id` or `task_id` as additional parameters.
note_format	Optional	String	The format of the note. Valid options are html or markdown (default).
artifact_id	Dependent	Integer	ID of the artifact for which you want notes.
task_id	Dependent	Integer	ID of the task for which you want notes.
title	Optional	String	Title for the note.
content	Optional	String	Content for the note.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.add_note API.

```
import phantom.rules as phantom

def on_start(container):

      phantom.debug('phantom.add_note')

      success, message, note_id = phantom.add_note(container=container, note_type='general', title='This is
a title.', content='This is the note body.')

      phantom.debug('phantom.add_note results: success {}, message {}, note_id {}'\
                    .format(success, message, note_id))
```

add_tags

Use the add_tags API to add tags to a container. On completion, the call returns a success flag and any response messages.

The add_tags API is supported from within a custom function.

```
phantom.add_tags(container=None, tags=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
tags	Required	String(s)	The tags to be added.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.add_tags API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.add_tags start')

    success, message = phantom.add_tags(tags=['tag1', 'tag2'])

    phantom.debug(
        'phantom.add_tags results: success: {}, message: {}'\
        .format(success, message)
    )

    return
```

add_task

Use the add_task API to add a task to a container. It returns a success flag, a success or error message, and the ID of the created task. The add_task API is supported from within a custom function.

```
phantom.add_task(container=None, name=None, owner=None, role=None, trace=False)
```

Parameter	Required?	Type	Description
container	Required	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object.
name	Optional	String	Name of the task to create.
owner	Optional	Integer or string	Identifier you can associate with the task.
role	Optional	Integer or string	Identifier for the role to associate with the task.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.add_task() API.

```
import phantom.rules as phantom

def on_start(container):
```

```
    phantom.debug('phantom.add_task')

    success, message, task_id = phantom.add_task(container=container, name='task name')

    phantom.debug('phantom.add_task results: success {}, message {}, task_id {}'\
                  .format(success, message, task_id))
```

add_workbook

Use the add_workbook API to add a workbook to a container. This API returns a two-tuple of a success flag and a message. The success flag is set to true if the workbook was successfully added, otherwise it is set to false. The message originates from the HTTP response from your Splunk SOAR (On-premises) web server.

The add_workbook API is supported from within a custom function.

```
def add_workbook(container=None, workbook_id=None, trace=False)
```

Parameter	Required?	Type	Description
container	Yes	Dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
workbook_id	Yes	Integer	The ID of the workbook to be added.
trace	No	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.add_workbook API.

```
import phantom.rules as phantom
def on_start(container):
    phantom.debug('phantom.add_workbook start')
    phishing_workbook_id = 4
   success, message = phantom.add_tags(workbook_id=phishing_workbook_id)
   success, message = phantom.add_workbook(workbook_id=phishing_workbook_id)
    if success:
        phantom.debug('phantom.add_workbook succeeded. API message: {}'.format(message))
        # Call on_success callback
    else:
        phantom.debug('phantom.add_workbook failed. API message: {}'.format(message))
        # Call on_fail callback
    return
```

close

The close API allows playbooks to close the specified container by setting its status to the default status of Resolved.

The close API is supported from within a custom function.

If using the save_object() API with the auto_delete value set to True, and the container_id value specified, the stored data is deleted when this container is closed or resolved.

```
phantom.close(container)
```

Parameter	Required?	Type	Description
container	Required	Dictionary	A container object.

This sample uses the phantom.close() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.close start')

    success, message = phantom.close(container)

    phantom.debug('phantom.close results: success: {}, message: {}, '\
                  .format(success, message))

    return
```

comment

Use the comment API to add a text comment to the container. On completion, the API returns a tuple of a success flag and any response messages.

The comment API is supported from within a custom function.

```
phantom.comment(container=None, comment=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
comment	Required	String	The comment to add to the container.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.comment() API.

```
import phantom.rules as phantom

def on_start(container):

    success, message = phantom.comment(comment='Example comment.')

    phantom.debug(
        'phantom.comment results: success: {}, message: {}' \
        .format(success, message)
    )

    return
```

create_container

Use the create_container API to create a new container. It returns a tuple of a success flag, any response message, and the numeric ID of the created container. If no container was created, the ID returned is None. Once a container is created, it is possible to retrieve container data using `get_container` by passing the created `container_id`.

The create_container API is supported from within a custom function.

```
phantom.create_container(name=None, label=None, container_type='default',
    template=None, trace=False, tenant_id=1)
```

Parameter	Required?	Type	Description

name	Required	String	The name of the case.
label	Required	String	The label of the case.
container_type	Optional	String	The type of container to create, either default or case.
template	Optional	Integer or String	Applies to containers of type case, and indicates the template to use on creation. If the container type is case and no template is provided, the default template is used. If no default template is set, an error is returned.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
tenant_id	Optional	Integer or String	The Splunk SOAR (On-premises) tenant ID as per the /rest/tenant API. This field is required if multi-tenancy is enabled.

This sample uses the phantom.create_container() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.create_container start')

    success, message, container_id = phantom.create_container(name='Example Name',
                                                              label='Example Label',
                                                              container_type='case')

    phantom.debug(
        'phantom.create_container results: success: {}, message: {}, container_id: {}'\
        .format(success, message, container_id)
    )

    return
```

delete_artifact

The delete_artifact API allows artifacts to be deleted as identified by an artifact ID. On completion, this API returns a Boolean value indicating whether the operation was successful.

> Prior to Splunk Phantom 3.0.x, only manually created artifacts were allowed to be edited or deleted. Splunk Phantom 3.0.x and later does not have this restriction, but requires that the user, either default or automation, have the delete container permission.

The delete_artifact API is supported from within a custom function.

```
phantom.delete_artifact(artifact_id=None)
```

Parameter	Required?	Type	Description
artifact_id	Required	Integer	The numerical ID of the artifact to be removed.

This sample uses the phantom.delete_artifact API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.delete_artifact start')
```

```
    success = phantom.delete_artifact(artifact_id=1)

    phantom.debug('phantom.delete_artifact results: success: {} '\
                  .format(success))

    return
```

delete_pin

The delete_pin API is used to delete an existing pin. On completion, this API returns a success flag and a message.

The delete_pin API is supported from within a custom function.

```
phantom.delete_pin(pin_id=None)
```

Parameter	Required?	Type	Description
pin_id	Required	Integer	The ID of the pin to be deleted.

Example usage to delete a pin.

```
def on_start(container):

    phantom.debug('phantom.delete_pin start')

    success, message = phantom.delete_pin(pin_id=3)

    phantom.debug(
        'phantom.delete_pin results:'\
        'success: {}, message: {}'.format(success, message)
    )

    return
```

get_notes

The get_notes API is used to get all the notes for the requested object, either a container, a task, or an artifact. On completion, this API returns the requested notes.

The get_notes API is supported from within a custom function.

```
phantom.get_notes(container=None, artifact_id=None, task_id=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. This container can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
artifact_id	Optional	Integer	ID of the artifact you want notes for.
task_id	Optional	Integer	ID of the task you want notes for.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.get_notes() API.

Example:

```
import phantom.rules as phantom
```

```
def on_start(container):

      phantom.debug('phantom.get_notes')

      for note in phantom.get_notes(container=container):
              phantom.debug('phantom.get_notes results: success {}, message {}, data {}'\
                            .format(note['success'], note['message'], note['data']))
```

get_phase

Use the get_phase API to retrieve the current phase of the container. On completion, this API returns a tuple of a success flag, any response messages, the phase ID and the phase name.

The get_phase API is supported from within a custom function.

```
phantom.get_phase(container=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.get_phase() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.get_phase start')

    success, message, phase_id, phase_name = phantom.get_phase()

    phantom.debug(
        'phantom.get_phase results: success: {}, message: {}, phase_id: {}, phase_name: {}'\
        .format(success, message, phase_id, phase_name)
    )

    return
```

get_tags

Use the get_tags API to retrieve the tags applied to a container. On completion, this API returns a tuple of a success flag, a response message, and a list of tags.

The get_tags API is supported from within a custom function.

```
phantom.get_tags(container=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.get_tags() API.

Example:

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.get_tags start')

    success, message, tags = phantom.get_tags()

    phantom.debug(
        'phantom.get_tags results: success: {}, message: {}, tags: {}'\
        .format(success, message, tags)
    )

    return
```

get_tasks

Use the get_tasks API to get all the tasks associated with a container. This API returns a tuple of a success flag (Boolean), any response message (string), and data (dictionary). The get_tasks API is supported from within a custom function.

```
phantom.get_tasks(container=None, trace=False)
```

Parameter	Required?	Type	Description
container	Required	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.get_tasks() API.

Example:

```
import phantom.rules as phantom

def on_start(container):

      phantom.debug('phantom.get_tasks')

      for task in phantom.get_tasks(container=container):
              phantom.debug('phantom.get_tasks results: success {}, message {}, data {}'\
                            .format(task['success'], task['message'], task['data']))
```

merge

Use the merge API to add items to a case. The merge API returns a tuple of a success flag and any response messages.

The merge API is supported from within a custom function.

```
 phantom.merge(case=None,
              container_id=None,
              artifact_id=None,
              ioc_field=None,
              playbook_run_id=None,
              action_run_id=None,
```

```
                vault_id=None,
                note_title=None,
                note_description=None,
                trace=False)
```

Parameter	Required?	Type	Description
case	Optional	Integer or dictionary	The case to add items to. This parameter can either be a numerical ID or a container object. If no case is provided, the currently running container is used.
container_id	Optional	Integer	The numerical ID of a container to add. All artifacts, playbook runs, action runs, vault items, notes and comments are copied. Mapping entries are created for each item that is copied, and all copied artifacts have their in_case flag set to true on the source artifact. If the container ID has already been added to the case, an error is returned. Any playbook or action runs that are running are not copied.
artifact_id	Optional	Integer	The numerical ID of an artifact to add. This parameter copies the artifact to the case, creates a mapping entry and sets the in_case flag to true on the source container. If the artifact ID has already been added to the case, an error is returned.
ioc_field	Optional	String	Common Event Format (CEF) key of the indicators of compromise (IOC) to add. Requires artifact ID to be specified. Copies a specific IOC from the specified artifact. A new artifact is created in the case containing the selected IOC and a mapping entry. If the IOC and artifact pair has already been added to the case, an error is returned.
playbook_run_id	Optional	Integer	The numerical ID of the playbook run to add. Copies the playbook run, playbook run logs and all playbook action runs to the case. Mapping entries are created for the playbook runs and action runs. If the playbook run has already been added to the case, an error is returned. Playbook runs that are 'running' cannot be added to a case.
action_run_id	Optional	Integer	The numerical ID of an action run, which can be retrieved from an action object or by using the get_summary API, which enumerates all the actions that were executed in the playbook. This parameter copies the action run to the case and creates a mapping entry. If the action run has already been added to the case, an error is returned. Action runs that are running cannot be added to a case.
vault_id	Optional	int	The numerical value of the vault file's ID attribute. Copies a file in the vault to the case. If the vault file has already been added, an error is returned.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.
note_title	Optional	String	The title of the note.
note_description	Optional	String	The description of the note.

Specify only one item to merge per phantom.merge call, except in the case of the artifact_id and ioc_field. Specifying multiple parameters to merge at once can cause unpredictable results.

This sample uses the phantom.merge API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.merge start')

    success, message = phantom.merge(artifact_id=1)

    phantom.debug('phantom.merge results: success {}, message: {}'\
                  .format(success, message))

    return
```

pin

Use the pin API to pin data to a container. Once complete, the API returns a tuple of a success flag, any response messages, and the pin ID.

The pin API is supported from within a custom function.

```
phantom.pin(container=None, message=None,
            data=None, pin_type=None,
            pin_style=None, truncate=True,
            name=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
message	Optional, unless the `data` parameter is not provided	String	A message associated with the pinned data.
data	Optional, unless the `message` parameter is not provided	String	The data to be pinned.
pin_type	Optional	String	The type of pin to create. The default pin type is 'card': card, data.
pin_style	Optional	String	The style of the pin. The default pin style type is 'grey': grey, blue, red.
truncate	Optional	Boolean	If the message or data fields are longer than the character limit, this flag determines if the data is automatically truncated or if an error is raised. The default is truncate=True, and the flag automatically truncates the message to the allowable character limit. truncate=False raises an error and no pin is created.
name	Optional	String	The name for the pin. If there is a name, it is unique in a container. If you try to create a named pin on a container where a pin with that name already exists, it updates that pin instead of creating a new one. This parameter lets you use the pin API to create or update a pin without needing to keep track of the pin ID.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.pin() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.pin start')

    success, message, pin_id = phantom.pin(container=container,
                                           data='192.168.1.7',
                                           message='This is a malicious IP',
                                           pin_type='card_small',
                                           pin_style='white')

    phantom.debug('phantom.pin results: success: {}, message: {}, '\
                  'pin_id: {}'.format(success, message, pin_id))

    return
```

promote

The promote API is used to promote a container to a case. The promote API is supported from within a custom function.

```
phantom.promote(container=None, template=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
template	Optional	Integer or string	Either a numerical ID or the name of the template to use for the case. If no template is provided, the default template is used. If no default is set, an error is returned.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.promote() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.promote start')

    # success, message = phantom.promote(template='Example Template Name')

    success, message = phantom.promote()

    phantom.debug(
        'phantom.promote results: success: {}, message: {}'\
        .format(success, message)
    )

    return
```

remove_tags

The remove_tags API is used to remove tags from a container. Upon completion, the API returns a success flag and any response messages. The remove_tags API is supported from within a custom function.

```
phantom.remove_tags(container=None, tags=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
tags	Required	String(s)	A string or list of strings that contains the tags to be removed.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.remove_tags() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.remove_tags start')

    success, message = phantom.remove_tags(tags=['tag1', 'tag2'])
```

```
    phantom.debug(
        'phantom.remove_tags results: success: {}, message: {}'\
        .format(success, message)
    )

    return
```

set_duetime

Use the set_duetime API to modify the due time of a container. This parameter returns a tuple of a success flag, a message if available, and the new due time.

The set_duetime API is supported from within a custom function.

```
phantom.set_duetime(container=None, operation='+', minutes=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
operation	Optional	String	A choice of add or + or subtract -, defaults to +.
minutes	Required	Integer	The number of minutes to add or subtract.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.set_duetime() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.set_duetime start')

    success, message, new_due_time = phantom.set_duetime(minutes=180)

    phantom.debug(
        'phantom.set_duetime results: success: {}, message: {}, new_due_time: {}'
        .format(success, message, new_due_time)
    )

    return
```

set_label

The set_label API allows users to dynamically change the label of a container. The label must already exist in system settings to be applied to a container. Upon completion, the API returns a tuple of a success flag and any response messages.

The set_label API is supported from within a custom function.

```
phantom.set_label(container=None, label=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. It can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
label	Required	String	A string containing the label to apply.

Parameter	Required?	Type	Description
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.set_label API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.set_label start')

    success, message = phantom.set_label(label='Example label')

    phantom.debug(
        'phantom.set_label results: success: {}, message: {}'\
        .format(success, message)
    )

    return
```

set_owner

Use the set_owner API to dynamically assign a container or task to a user or role. Upon completion, the API returns a tuple of a success flag and a message, if available.

The set_owner API is supported from within a custom function.

```
phantom.set_owner(container=None, task_id=Int, user='', role='', trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The ID of the container assigned ownership. If no container is provided, the currently running container is used.
task_id	Optional	Integer	The task ID of the task assigned ownership. If an empty string is sent, this API removes any owners from the task.
user	Optional	String or integer	Valid Splunk SOAR (On-premises) username or ID assigned, or an empty string to remove assignments on the container. If you don't set a user, or you send an empty string, you remove assignments from the container.
role	Optional	Integer or string	The name or ID of the role. If no user or role values are set or if empty strings are sent, role assignments are removed from the container or task.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.set_owner() API.

```
import phantom.rules as phantom

def on_start(container):

  # assign container to user@company.com
  success, message = phantom.set_owner(container=container, user='user@company.com')
  phantom.debug(
      'phantom.set_owner results: success: {}, message: {}'
      .format(success, message)
  )
```

```
  # container unassigned, no user assigned
  success, message = phantom.set_owner(container=container, user="")
  phantom.debug(
      'phantom.set_owner results: success: {}, message: {}'
      .format(success, message)
  )
  return
```

set_phase

Use the set_phase API to set the current phase of the container. This parameter returns a tuple of a success flag and a message, if available.

The set_phase API is supported from within a custom function.

```
phantom.set_phase(container=None, phase=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or Dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
phase	Required	String or Integer	Use either a phase name or a numerical ID for the phase to apply.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.set_phase API.

Example:

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.set_phase start')

    success, message = phantom.set_phase(phase='Example Phase')

    phantom.debug(
        'phantom.set_phase results: success: {}, message: {}'\
        .format(success, message)
    )

    return
```

set_sensitivity

Use the set_sensitivity API to dynamically change the sensitivity of a container while it is being processed. The system supports four levels of sensitivity in accordance to the US-CERT Traffic Light Protocol. The levels supported are red, amber, green, and white.

The set_sensitivity API is supported from within a custom function.

```
phantom.set_sensitivity(container, sensitivity)
```

Parameter	Required?	Type	Description
container	Required	Dictionary	The current container object.

Parameter	Required?	Type	Description
sensitivity	Required	String	The desired sensitivity. Options include red, amber, green, and white.

set_severity

Use the set_severity API to dynamically change the severity of a container while it is being processed.

The set_severity API is supported from within a custom function.

```
phantom.set_severity(container, severity)
```

Parameter	Required?	Type	Description
container	Required	Dictionary	The current container object.
severity	Required	String	The name of your desired severity level.

set_status

Use the set_status API to update the status of a container. Returns a tuple of a success flag and a message, if available.

The set_status API is supported from within a custom function.

```
phantom.set_status(container=None, status=None, trace=False)
```

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
status	Required	String	Set the status of either new, open, resolved, closed, or a custom status defined by an administrator.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

Prior to Splunk Phantom 4.5 the statuses of Resolved and Closed were interchangeable. With customizable statuses, the statuses are no longer interchangeable. For backwards compatibility, if there is no active status named Resolved, calling this API with a value of resolved causes the set_status API to retry with a value of closed. If you depend on resolved and closed being interchangeable, call the phantom.set_status() API with status="closed" instead.

This sample uses the phantom.set_status API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.set_status start')

    success, message = phantom.set_status(status='open')

    phantom.debug(
        'phantom.set_status results: success: {}, message: {}'\
        .format(success, message)
    )

    return
```

update

Use the update API to make an update to a container. It returns a tuple of a success flag and any response messages.

The update API is supported from within a custom function.

```
phantom.update(container, update_dict)
```

Parameter	Required?	Description
container	Required	The container object to be updated.
update_dict	Required	A JSON serializable Python dictionary which contains updates to the container. Include only the fields that you want to change.

Example usage to change the container label to email and sensitivity to red.

```
update_data = { "label": "email", "sensitivity": "red" }
success, message = phantom.update(container, update_data)
```

Example usage to append a new tag to the container.

```
current_tags = phantom.get_container(container['id'])['tags']
current_tags.append('newtag2')
update_data = { "tags" : current_tags }
success, message = phantom.update(container, update_data)
```

Changing the label on a container can potentially trigger automatic playbook runs on that container if there are any playbooks set to active that run on that container label. Standard automatic playbook run constraints apply, such as actions not running if the playbook had already run and there are no new artifacts, or if the container is in a closed state. You can set "run_automation": false in the JSON to block playbooks running on this update call.

Updating and reading custom field values

For information on updating and retrieving custom field values with the container.update API, refer to Update and read custom field values.

update_pin

Use the update_pin API to update an existing pin. Upon completion, this API returns a success flag and a message.

The update_pin API is supported from within a custom function.

```
phantom.update_pin(pin_id=None, message=None, data=None, pin_type=None, pin_style=None)
```

Parameter	Required?	Type	Description
pin_id	Required	Integer	The ID of the pin to be updated.
message	Optional	String	A new message string for the pin.
data	Optional	String	A new data string for the pin.
pin_type	Optional	String	A new type for the pin.
pin_style	Optional	String	A new style for the pin.

Parameter	Required?	Type	Description

Use named pins instead of this API to handle updating pins.

In this example, an existing pin's message and data is updated.

```
def on_start(container):

    phantom.debug('phantom.update_pin start')

    success, message = phantom.update_pin(pin_id=3, message="A new message",
                                          data="Some new data")

    phantom.debug(
        'phantom.update_pin results:'\
        'success: {}, message: {}'.format(success, message)
    )

    return
```

Data management automation API

The Splunk SOAR (On-premises) Automation API allows security operations teams to develop detailed and precise automation strategies. Playbooks can serve many purposes, ranging from automating minimal investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of data management automation using playbooks.

add_list

Use the add_list API to append a new row of data to the custom list named by list name. If the list does not exist, it is created. Values are converted to a list of strings. Upon completion, the API returns a tuple of a success flag and response messages.

The add_list API is supported from within a custom function.

```
phantom.add_list(list_name=None, values=None)
```

Parameter	Required?	Type	Description
list_name	Required	String	The name of the custom list to add an item to.
values	Required	String or list of strings	The values to be added.

This sample uses the phantom.add_list API.

```
import phantom.rules as phantom
import json

def on_start(container):
    #in the product in 'Playbooks / Custom Lists', define a
    # list called 'executives' and then access it here
    success, message = phantom.add_list(
        list_name='executives', values=[ 'bob.jones@splunk.com' ] )

    phantom.debug(
        'phantom.add_list results: success: {}, message: {}' \
```

```
        .format(success, message)
    )

    success, message = phantom.add_list(
        list_name='executives', values=[ 'susan.smith@splunk.com' ] )

    phantom.debug(
        'phantom.add_list results: success: {}, message: {}' \
        .format(success, message)
    )

    return

def on_finish(container, summary):
    return
```

check_list

Use the check_list API to check whether a value is in a custom list or not. The default behavior, case_sensitive=False, is case insensitive and searches complete strings using substring=False. The API returns a tuple of a success flag, any response messages, and the number of matching rows in the custom list.

The check_list API is not supported from within a custom function.

```
phantom.check_list(list_name=None, value=None, case_sensitive=False, substring=False)
```

Parameter	Required?	Type	Description
list_name	Required	String	The name of the custom list to be searched.
value	Required	String or list of strings	The value to be searched.
case_sensitive	Optional	Boolean	Change the case sensitivity with this parameter. The default behavior is case insensitive.
substring	Optional	Boolean	Change the substring with this parameter. The default behavior is complete string match.

This sample uses the phantom.check_list() API.

```
import phantom.rules as phantom

def on_start(container):
    phantom.debug('phantom.check_list start')

    success, message, matched_row_count = \
        phantom.check_list(list_name='Example List', value='Example Value')

    phantom.debug(
      'phantom.check_list results: success: {}, message: {}, matched_row_count: {}'\
      .format(success, message, matched_row_count)
    )

    return

def on_finish(container, summary):
    return
```

clear_data

Use the clear_data API to clear data stored from the phantom.save_data method and delete it from the persistent store.

The clear_data API is not supported from within a custom function.

```
phantom.clear_data(key)
```

Parameter	Required?	Type	Description
key	Required	String	The key provided to or by the save_data API. For more information, see save_data.

clear_object

Use the clear_object API to delete data saved through the save_object() API.

The clear_object API is not supported from within a custom function.

```
phantom.clear_object(key=None, container_id=None, playbook_name=None, repo_name=None)
```

Parameter	Required?	Description
key	Required	The key parameter that was used in the save_object() API.
container_id	Optional, unless `playbook_name` is not provided. Can also be used together with a `playbook_name`.	The same `container_id` parameter that was used in save_object() API.
playbook_name	Optional, unless `container_id` is not provided. Can also be used together with `container_id`.	The same `playbook_name` parameter that was used in save_object() API.
repo_name	Optional	The same `repo_name` parameter that was used in save_object() API.

The `container_id` and `playbook_name` parameters support Postgres LIKE patterns. If the pattern does not contain percent signs or underscores, then the pattern only represents the string itself. In that case, LIKE acts like the equals operator. An underscore in the pattern stands for any single character and a percent sign matches any sequence of zero or more characters.

delete_from_list

Use the delete_from_list API to remove rows from a list that contain a specific value. The delete_from_list API returns a tuple of a success flag and any response messages.

The delete_from_list API is supported from within a custom function.

```
phantom.delete_from_list(list_name=None, value=None, column=None, remove_all=False, remove_row=False)
```

Parameter	Required?	Type	Description
list_name	Required	String	The name of the custom list to be modified.
value	Required	String	Replaces cells containing value with None.
column	Optional	Positive integer	Zero-based index, checks for values in this column.
remove_all	Optional	boolean	If True, replace all occurrences of value with None. Otherwise, the API fails if multiple are found.
remove_row	Optional	boolean	If True, remove the full row where value was found.

The following sample uses the phantom.delete_from_list() API.

```
import phantom.rules as phantom
import json

def on_start(container):
    #in the product in 'Playbooks / Custom Lists', define a
    # list called 'example' with two rows. Creates the list if it does not exist.
    success, message = phantom.delete_from_list(
        list_name='example',
        value='deleteme')
    phantom.debug(
      'phantom.delete_from_list results: success: {}, message: {}'\
      .format(success, message)
    )
    return

def on_finish(container, summary):
    return
```

You can't delete all of the rows from the list, and phantom.delete_from_list commands that attempt to do so result in errors. At least one row must be present in the list.

get_data

Use the get_data API to return data stored from the phantom.save_data method and delete data from the persistent store if the clear_data parameter is set to the default of True.

The get_data API is not supported from within a custom function.

```
phantom.get_data(key, clear_data=True)
```

Parameter	Required?	Type	Description
key	Required	String	The key provided to or by the save_data API. For more information, see save_data.

Only JSON-compliant objects, dictionaries, lists, strings, and numbers are supported as objects that can be saved and retrieved.

get_list

Use the get_list API to access custom lists. Custom lists are lists of dictionaries that allow you to manage data that can be referenced in Splunk SOAR (On-premises) playbooks. You can view or maintain these lists from navigation to the Splunk SOAR (On-premises) main menu, and then click **Playbooks > Custom Lists**.

The get_list API is supported from within a custom function.

```
phantom.get_list(list_name=None, values=None, column_index=-1, trace=False)
```

Parameter	Required?	Type	Description
list_name	Required	String	The name of the custom list to retrieve.
values	Optional	String or list of strings	A value or a list of values to search. If a value isn't included, the full list is retrieved.
column_index	Optional	Integer	Used to specify a specific column to retrieve.
trace	Optional	Boolean	

Parameter	Required?	Type	Description
			Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.get_list() API.

```
import phantom.rules as phantom
import json

def on_start(container):
    #in the product in 'Playbooks / Custom Lists', define a
    # list called 'executives' and then access it here
    success, message, execs = phantom.get_list(list_name='executives')

    phantom.debug(
        'phantom.get_list results: success: {}, message: {}, execs: {}'\
        .format(success, message, execs)
    )

    return

def on_finish(container, summary):
    return
```

get_object

Use the get_object API to retrieve data that was saved through the save_object API. See save_object for a sample playbook and usage of this API.

The get_object API is not supported from within a custom function.

```
phantom.get_object(key=None, clear_data=False, container_id=None, playbook_name=None, repo_name=None)
```

Parameter	Required?	Description
key	Required	The key specified in the save_object() API that is used when saving data.
clear_data	Optional	If set True, this parameter clears the data after fetching. Defaults to False.
container_id	Optional, unless playbook_name is not specified. Can also be specified with a `playbook_name`.	The container ID specified when the data was saved.
playbook_name	Optional, unless container_id is not specified. Can also be specified with container_id	The playbook name as specified when the data was saved.
repo_name	Optional	The repository name as specified when saving the data in the save_object API.

The key parameter supports Postgres LIKE patterns. If the pattern does not contain percent signs or underscores, then the pattern only represents the string itself. In that case, LIKE acts like the equals operator. An underscore in the pattern stands for any single character and a percent sign matches any sequence of zero or more characters.

This sample shows the return value as a list of dictionaries where each dictionary has search criteria and the value for the specified combination of parameters.

```
[
    {
```

```
        "composite_key": {
            "container_id": <>,
            "playbook_name": <>,
            "key": <>
        },
        "value": {<>
        }
    }
]
```

get_run_data

Use the get_run_data method to return the data value saved for the specified key through the phantom.save_run_data() method. If the key is not specified, the get_run_data API returns data for all the keys as a string object.

The get_run_data API is supported from within a custom function.

```
phantom.get_run_data(key=None)
```

This sample shows result data if a key is not specified when calling get_run_data.

```
{
   "specified_key_on_save_run_data_call" : {
      "auto_" : true,
      "data_" : "specified value on save run data call with key provided"
   },
   "ef106ce1-301d-490d-9b96-d16b7e3a1a85" : {
      "auto_" : true,
      "data_" : "specified value on save run data call without key provided"
   }
}
```

remove_list

Use the remove_list API to delete a list. If you use the empty_list option, the list still exists, but is cleared of all values.

The remove_list API is supported from within a custom function.

```
phantom.remove_list(list_name=None, empty_list=False, trace=False)
```

Parameter	Required?	Type	Description
list_name	Required	String	The name of the custom list to delete.
empty_list	Optional	Boolean	Setting this parameter to True clears the list contents instead of removing the list.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

For remove_list API to work, give the automation user permissions to delete lists.

This sample uses the phantom.remove_list() API.

```
import phantom.rules as phantom
import json

def on_start(container):
    phantom.debug('phantom.remove_list start')

    success, message = phantom.remove_list(list_name='example')
```

```
    phantom.debug(
      'phantom.remove_list results: success: {}, message: {}'\
      .format(success, message)
    )

    return

def on_finish(container, summary):
    return
```

save_data

Use the save_data API to save the data provided in the value parameter and to return a generated key that can be used to retrieve data. This data can be saved and retrieved within or across playbooks when you provide a matching key. To save and get data across playbook runs, select a fixed name for the key. Only JSON compliant objects, dictionaries, lists, strings, and numbers are supported as objects that can be saved and retrieved.

The save_data API is not supported from within a custom function.

```
phantom.save_data(value, key=None)
```

save_object

Use the save_object API to save data by key, container ID or the playbook name to be retrieved when executing playbooks on containers. You can save a key and value pair along with the context of container ID or the playbook name. Only JSON compliant objects, dictionaries, lists, strings, and numbers are supported as objects that can be saved and retrieved.

The save_object API is not supported from within a custom function.

```
 phantom.save_object(key=None, value=None, container_id=None,
                     auto_delete=False, playbook_name=None,
                     repo_name=None)
```

Parameter	Required?	Description
key	Required	Specify key to save and retrieve data by this unique key.
value	Required	The data to be saved. Use a expected to be a Python dictionary object.
container_id	Optional, unless the playbook_name parameter is not specified.	This parameter is the container ID as a context to the data being saved. You must provide this parameter if auto_delete is True.
auto_delete	Optional, unless the container_ID parameter is not provided	Defaults to True. If set to True, the data is deleted when the container is closed. You can use the clear_object parameter to delete the data. If the parameter is set True, you must provide the container ID.
playbook_name	Optional	The playbook name, which is also saved as context to the data.
repo_name	Optional	Specify this parameter when a playbook name is provided, because a particular playbook might exist in more than one repository.

This sample playbook uses save_object.

```
import phantom.rules as phantom
import json
from datetime import datetime, timedelta
```

```
def on_start(container):
    phantom.debug('on_start() called {}'.format(container))

    pb_info = phantom.get_playbook_info()
    phantom.debug(pb_info)
    if not pb_info:
        return

    playbook_name = pb_info[0].get('name', None)
    container_id = container['id']


    # SAVE data with key, container id and Playbook name
    phantom.save_object(key="key1", value={'value':'key 1 data for
        container and playbook'}, auto_delete=True, container_id = container_id,
        playbook_name=playbook_name)

    phantom.save_object(key="key2", value={'value':'key 2 data for
        container and playbook'}, auto_delete=True, container_id = container_id,
        playbook_name=playbook_name)

    # SAVE data with key, container id but NO Playbook name
    phantom.save_object(key="key1", value={'value':'key 1 data for
        only container and NO playbook'}, auto_delete=True,
        container_id = container_id)

    # SAVE data with key, Playbook name and NO container id
    phantom.save_object(key="key1", value={'value':'key 1 data for
        only playbook and not container'}, auto_delete=False,
        playbook_name=playbook_name)


    my_key = "key1"
    data = phantom.get_object(key=my_key,
            container_id = container_id, playbook_name=playbook_name)

    phantom.debug("get by key, container_id and playbook name:
        {} records found".format(len(data)))
    phantom.debug(data)

    data = phantom.get_object(key=my_key, container_id = container_id)

    phantom.debug("get by key, and container_id and
        NO playbook name: {} records found".format(len(data)))
    phantom.debug(data)

    data = phantom.get_object(key=my_key,
        playbook_name=playbook_name)

    phantom.debug("get by key, and playbook name and no container id:
        {} records found".format(len(data)))
    phantom.debug(data)

    data = phantom.get_object(key=my_key, playbook_name="%%")

    phantom.debug("get by key, and ALL playbook name and
        no container id: {} records found".format(len(data)))
    phantom.debug(data)

    data = phantom.get_object(key="%%",
        container_id = container_id)
```

```
    phantom.debug("get for ALL key, and container_id and
        no playbook name: {} records found".format(len(data)))
    phantom.debug(data)

    data = phantom.get_object(key="%%", container_id = container_id,
        playbook_name=playbook_name)

    phantom.debug("get for ALL key, and container_id and playbook name:
        {} records found".format(len(data)))
    phantom.debug(data)

    # TESTING CLEAR API

    data = phantom.get_object(key="key1", container_id = container_id,
        playbook_name=playbook_name)
    phantom.debug("BEFORE clear ... get for key1, and container_id and playbook
        name: {} records found".format(len(data)))
    phantom.debug(data)

    phantom.clear_object(key="key1", container_id = container_id,
        playbook_name=playbook_name)

    data = phantom.get_object(key="key1", container_id = container_id,
        playbook_name=playbook_name)

    phantom.debug("AFTER clear ... get for key1, and container_id and playbook
        name: {} records found".format(len(data)))
    phantom.debug(data)


    return

def on_finish(container, summary):
    phantom.debug('on_finish() called')
    return
```

The output of the playbook in debugger shows the following results.

```
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): Starting playbook 'local/automation_data' (id: 112, version: 35) on
'incident'(id: 10) with playbook run id: 72
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): calling on_start() on incident 'CryptoLocker Ransomware Infection
(new, SLA breached) (192.168.1.41)'(id: 10).
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): on_start() called {'sensitivity': 'amber', 'create_time':
'2017-03-23 03:54:12.981519+00', 'owner': 'bob.tailor@splunk.com', 'id': 10, 'close_time': '', 'severity':
'high', 'label': 'incident', 'due_time': '2017-03-18 21:34:47.217516+00', 'version': '1',
'current_rule_run_id': 72, 'status': 'new', 'owner_name': '', 'hash': 'f407a85b849baecdb34d27da1e1431dc',
'description': 'CryptoLocker has been detected on finance system 192.168.1.41 running on ESXi server
192.168.1.40', 'tags': [], 'start_time': '2014-09-04 14:40:33+00', 'asset_name': 'qradar_entr',
'artifact_update_time': '2017-03-23 03:54:15.990076+00', 'container_update_time': '', 'kill_chain': '',
'name': 'CryptoLocker Ransomware Infection (new, SLA breached) (192.168.1.41)', 'ingest_app_id': '',
'source_data_identifier': '45', 'end_time': '', 'artifact_count': 2}
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
    {
        "parent_playbook_run_id": "0",
        "name": "automation_data",
        "run_id": "72",
        "scope_artifacts": [],
        "scope": "new",
        "id": "112",
        "repo_name": "local"
    }
```

```
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): save_object()
called:key=key1,auto_delete=True,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): save_object()
called:key=key2,auto_delete=True,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): save_object()
called:key=key1,auto_delete=True,container_id=10,playbook_name=None,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): save_object()
called:key=key1,auto_delete=False,container_id=None,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=key1,
clear_data=False,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get by key, container_id and playbook name: 1 records found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
    {
        "composite_key": {
            "container_id": 10,
            "playbook_name": "automation_data",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for container and playbook"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=key1,
clear_data=False,container_id=10,playbook_name=None,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get by key, and container_id and NO playbook name: 1 records found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
    {
        "composite_key": {
            "container_id": 10,
            "playbook_name": "",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for only container and NO playbook"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=key1,
clear_data=False,container_id=None,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get by key, and playbook name and no container id: 1 records found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
    {
        "composite_key": {
            "container_id": 0,
            "playbook_name": "automation_data",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for only playbook and not container"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=key1,
clear_data=False,container_id=None,playbook_name=%%,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get by key, and ALL playbook name and no container id: 1 records
found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
```

```
[
    {
        "composite_key": {
            "container_id": 0,
            "playbook_name": "automation_data",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for only playbook and not container"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=%%,
clear_data=False,container_id=10,playbook_name=None,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get for ALL key, and container_id and no playbook name: 1 records
found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
    {
        "composite_key": {
            "container_id": 10,
            "playbook_name": "",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for only container and NO playbook"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=%%,
clear_data=False,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get for ALL key, and container_id and playbook name: 2 records
found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
    {
        "composite_key": {
            "container_id": 10,
            "playbook_name": "automation_data",
            "key": "key2"
        },
        "value": {
            "value": "key 2 data for container and playbook"
        }
    },
    {
        "composite_key": {
            "container_id": 10,
            "playbook_name": "automation_data",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for container and playbook"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=key1,
clear_data=False,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): BEFORE clear ... get for key1, and container_id and playbook name:
1 records found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[
```

```
    {
        "composite_key": {
            "container_id": 10,
            "playbook_name": "automation_data",
            "key": "key1"
        },
        "value": {
            "value": "key 1 data for container and playbook"
        }
    }
]
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): clear_object()
called:key=key1,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): get_object() called:key=key1,
clear_data=False,container_id=10,playbook_name=automation_data,repo_name=None
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): AFTER clear ... get for key1, and container_id and playbook name: 0
records found
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):
[]

Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): No actions were executed
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT):

Playbook 'automation_data' (playbook id: 112) executed (playbook run id: 72) on incident 'CryptoLocker
Ransomware Infection (new, SLA breached) (192.168.1.41)'(container id: 10).
   Playbook execution status is 'success'
        Total actions executed: 0

Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): on_finish() called
Fri Mar 24 2017 07:13:08 GMT-0700 (PDT): {"message":"No actions were
executed","playbook_run_id":72,"result":[],"status":"success"}
```

If auto_delete is False, no container ID is provided, or the container isn't closed, the data isn't deleted and can waste space.

save_run_data

Use the save_run_data API to save the value in a key only in the context of the playbook run or execution. This data is automatically deleted when the playbook execution completes unless the auto parameter is set to False.

The save_run_data API is not supported from within a custom function.

```
phantom.save_run_data(value=None, key=None, auto=True)
```

set_action_limit

Use the set_action_limit API in your playbook's on_start() block to set the maximum number of action calls that can be executed. The default is 50 action calls per container per playbook. Each phantom.act() call can still result in multiple actions performed, resulting in more actions than this setting.

The set_action_limit API is not supported from within a custom function.

```
phantom.set_action_limit(limit)
```

set_list

Use the set_list API to replace the contents of a list. This API returns a tuple of a success flag and any response messages.

The set_list API is supported from within a custom function.

```
phantom.set_list(list_name=None, values=None)
```

Parameter	Required?	Type	Description
list_name	Required	String	The name of the custom list to modify.
values	Required	List of Lists	The values to set.

This sample uses the phantom.set_list() API.

```
import phantom.rules as phantom
import json

def on_start(container):
    #in the product in 'Playbooks / Custom Lists', define a
    # list called 'example' with two rows. Creates the list if it does not exist.
    success, message = phantom.set_list(
        list_name='example',
        values=[ ['a', 'list', 'of', 'values'], ['second', 'row'] ])
    phantom.debug(
      'phantom.set_list results: success: {}, message: {}'\
      .format(success, message)
    )
    return

def on_finish(container, summary):
    return
```

Data access automation API

The Splunk SOAR (On-premises) Automation API allows security operations teams to develop detailed and precise automation strategies. Playbooks can serve many purposes, ranging from automating minimal investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of data access using playbooks.

collect

Use the collect API to gather information from the associated artifacts of a container or action results that you get in the action callback or through the get_action_results() API. You can also use the collect API to obtain a listing of all IP addresses or all file hashes across all artifacts by specifying the appropriate data path into the artifact JSON. Or, extract all country ISO codes from the action results of action geolocate IP and pass the collect API into the results object. You can specify either one datapath as a string for the information you want to extract from action results, or you can specify more than one datapath in a list of datapath strings.

The collect API is supported from within a custom function.

```
phantom.collect(container, #this can be a container or an action results object
```

```
datapath,
scope='new',
limit=100,
none_if_first=False)
```

Parameter	Required	Description
container	Required	The container that is available to the user in on_start(), on_finish(), or any action callback. It can be a results object that you get in the action callback or through the get_action_results() API.

datapath	Required	The path of the element in the JSON schema to access or retrieve it from associated artifacts of a container or the action results object.
		
		The following are example datapaths for a container:
		
		1. Collect all file hashes from all the artifacts of a container: `phantom.collect(container, "artifact:*.cef.fileHash")`
		2. Collect all file hashes of a specific type (events) of a container: `phantom.collect(container, "artifact:events.cef.fileHash")`
		3. You can specify a substring to be searched across matching artifact types. The substring only applies to artifact type. `phantom.collect(container, "artifact:*event*.cef.fileHash")` This call finds file hashes across all the artifacts that have "event" as a substring.
		
		The following are example datapaths for action results:
		
		1. Extract longitude from the results of the geolocate IP action: `phantom.collect(results, "action_result.data.*.longitude")`
		2. Extract the number of positive detections from the results of the file reputation action using the VirusTotal app: `phantom.collect(results, "action_result.data.*.positives")`
		3. Extract three items from action results of the file reputation action using the Reversing Labs app:
		`def file_reputation_cb(action, success, container, results, handle):`
		
		`paths = ['action_result.data.*.status',`
		`'action_result.parameter.hash',`
		`'action_result.summary.positives']`
		
		`data = phantom.collect(results, paths)`
		
		`phantom.debug(data)`
		
		Example output:
		
		`[`
		`[`
		`"MALICIOUS",`
		`"70FEEC581CD97454A74A0D7C1D3183D1",`
		`26`
		`]`
		`]`
		
		If the datapath was specified as a string the result is a list, unlike the previous output.
		
		`data = phantom.collect(results, 'action_result.parameter.hash')`
		
		`phantom.debug(data)`
		Example output (list):

Parameter	Required	Description
		["70FEEC581CD97454A74A0D7C1D3183D1",] If you specify a list of datapaths for extracting data from action results, the results are formatted as a table, where each column represents the respective datapath. If you specify a single datapath as a string, Splunk SOAR returns the data corresponding to one column.

scopeOptionalThis parameter defines if the data has to be collected from artifacts and over what range of time the data is collected. The `scope` parameter can be new, which implies that the information has to be collected only from new artifacts since the playbook last ran on that container. The `all scope` parameter implies that the information has to be collected from all of the artifacts in the container.

An active playbook runs on a container after it has been created and every time new artifacts are added to the container. You can use the scope parameter when you want every instance of a playbook run to process only new artifacts that are added to the container. Every time you modify the playbook, it is considered a new playbook and it causes the playbook execution to start with all artifacts in the container until that instance in time. Then, the scope parameter collects only what has been added to the container after the previous instance of playbook execution.

limitOptionalThis parameter enforces the maximum number of artifacts that can be retrieved in this call. If the `limit` parameter is not specified, the limit is 2,000.none_if_firstOptionalWhen the collect API call is executed from a playbook for the first time on a container, even with the scope='new' argument, it collects all the artifacts since the container was created. Use this parameter to change the behavior of the collect API call executed for the first time from this playbook on a container. You can also use this parameter to specify whether the playbook collects all artifacts since the container was created, or only those artifacts added since the first time the playbook was executed on the container. Use 'True' for this parameter if you want the playbook to not get any existing artifacts the first time it is run on the container. Then, on subsequent playbook runs, it gets only the artifacts added since the first playbook run.

collect2

The collect2 API is an extension of the phantom.collect() API. It adds the `filter_artifacts` parameter, which is a list of artifacts whose values are returned. To learn more about the datapaths used in the collect2 API, see Understanding datapaths.

The collect2 API is supported from within a custom function.

```
phantom.collect2(container=None,
            action_results=None,
            action_name=None,
            datapath=None,
            filter_artifacts=None,
            tags=None,
            scope='new',
            limit=100,
            trace=False)
```

Parameter	Required?	Description
container	Required	The container dictionary object that is passed to the playbook across various functions.
action_results	Optional, unless the `action_name` parameter is not provided.	The action results passed into any callback function, or a subset of action results that are filtered from a phantom.condition() call. Results may be a mix of custom function and action results.
action_name	Optional, unless the `action_results` parameter is not provided.	The custom name specified for the action or custom function in the phantom.act() API. This parameter allows action results to be returned based on the action name or custom function name.

Parameter	Required?	Description
		If a block name is supplied to the collect2 API either from the datapath or the `action_name` keyword argument, then any action result that does not match the supplied name is ignored.
datapath	Required	A list of datapaths. A datapath is the path of the element in the JSON schema to be able to access or retrieve it from associated action results, custom function results, or artifacts. For more information, see collect.
filter_artifacts	Optional	IDs of artifacts returned from a phantom.condition() call.
tags	Optional	A list of tags used to filter artifacts.
scope	Optional	Scope of artifacts to retrieve. The default is new. See the phantom.collect() API for more details.
limit	Optional	The maximum number of results to be returned. For more information, see collect.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

Although literal values are valid datapaths, the collect2 API does not interpret them as such.

This sample uses the phantom.collect2() API.

Example request

```
collect2(
    container=container,
    action_results=[
        {
            'name': 'geolocate_ip_1',
            'action_results': [{
                'parameter': {
                    'ip': '10.0.0.1',
                },
            }],
           action_name,
        },
        {
            'name': 'geolocate_ip_1',
            'action_results': [{
                'parameter': {
                    'ip': '10.0.0.2',
                },
            }],
            'summary': {},
        },
    ],
    datapath=[
        'action_result.parameter.ip',
    ],
)
```

Beginning with Splunk Phantom 4.9, if the 'summary' key does not exist for an action result object, then 'summary' will default to an empty dict, for example 'summary': {}.

Example response

```
[
    ['10.0.0.1'],
    ['10.0.0.2'],
]
```

collect_from_contains

The collect_from_contains API functions similarly to collect, but instead of using datapaths for the values you want, you instead provide a contains value. This action returns a flat list of all the unique values that match at least one contains in the list. The call returns None if it fails.

The collect_from_contains API is supported from within a custom function.

```
phantom.collect_from_contains(container=None,
                              action_results=None,
                              contains=None,
                              tags=None,
                              scope=None,
                              filter_artifacts=None,
                              include_params=True,
                              limit=None,
                              trace=False)
```

Parameter	Required?	Description
container	Optional, unless the `action_results` parameter is not provided.	Passing this parameter searches for contains in the Common Event Format (CEF) values of that container.
action_results	Optional, unless the `container` parameter is not provided.	This parameter is an action result, like what is passed to a callback from phantom.act() as Result. Search for values matching the contains in this action result.
contains	Required	A list of contains to filter by.
tags	Optional	A list of tags used to further filter artifacts.
filter_artifacts	Optional	The IDs of artifacts that were returned from a phantom.condition()call.
include_params	Optional	If set to False, ignore values with matching contains if they are a parameter to an action. This value is only used if the action_result parameter is passed in.
scope	Optional	Scope of artifacts to retrieve. The default is new. This parameter is only used if a container is provided. For more information, see collect.
limit	Optional	Maximum number of artifacts to match. This value is used only if a container is provided.
trace	Optional	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

This sample uses the phantom.collect_from_contains() API.

```
import phantom.rules as phantom

def geolocate_ip(action, success, container, results, handle):

    # We have already created various artifacts for this event
    collected_ips = phantom.collect_from_contains(container=container, contains=["ip"])
    # [ "8.8.8.8", "8.8.4.4", "1.1.1.1", ... ]

    parameters = []
```

```
    for ip in collected_ips:
        parameters.append({
            'ip': ip
        })

    phantom.act("geolocate ip", parameters=parameters, app={ "name": "MaxMind" }, name="geolocate_ip")
    return

def collect_from_action_result(results):

    return phantom.collect_from_contains(action_results=results, contains=["url", "domain"])
```

get_action_results

Use the get_action_results API to retrieve the action results using the action JSON that was given in the action callback or the action run ID that was given in the action JSON. The API call get_summary() also returns one or more app run IDs that can be passed in as the optional parameter.

The get_action_results API is supported from within a custom function.

```
phantom.get_action_results(action=None,
                           action_run_id = 0,
                           app_run_id = 0,
                           result_data=True,
                           action_name=None,
                           playbook_run_id=0,
                           flatten=True)
```

Parameter	Required?	Description
action	Optional, unless the `action_run_id` and `app_run_id` parameters are not provided.	Action JSON object provided in the action callback. Using this parameter provides the action results from the action that completed and triggered the callback function.
action_run_id	Optional, unless the `action` and `app_run_id` parameters are not provided.	The ID of the action run. Use this parameter to obtain action results from any completed action runs from the current playbook. The action_run_id parameter can be obtained from the previously noted action JSON object or by calling the phantom.get_summary() API, which enumerates all of the actions that were executed in the playbook.
app_run_id	Optional, unless the `action` and `action_run_id` parameters are not provided.	The ID of the app run. This parameter can be obtained by calling the phantom.get_summary() API, which enumerates all of the actions that were executed in the playbook.
result_data	Optional	The default is True. If the user doesn't need to obtain the full action results or needs summary information, set this parameter to False.
action_name	Optional	The unique name provided to an action execution by using the phantom.act() parameter `name`.
playbook_run_id	Optional	The playbook run ID that uniquely identifies the playbook execution instance. A default value of zero implies the current playbook execution instance.
flatten	Optional	The default is True. An action can be executed on more than one asset and for many sets of parameters. Flattening provides a result dictionary object for each combination of asset and parameter, even if many parameters were used in a single action. Setting this variable to False generates results as provided in action callbacks or when viewing the action results in Investigation widgets.

A single phantom.act() API call can be executed on multiple sets of parameters on more than one asset. Each instance of phantom.act() call is identified by a unique action run ID. One action execution on each asset results in a corresponding app execution, each of which is identified by a unique app run ID Parameters of an action execution on each app, on their

respective assets, can be part of the same app run.

This sample uses the phantom.get_action_results() API.

```
import phantom.rules as phantom
import json

def collect_params(container, datapath, key_name):
    params = []
    items = set(phantom.collect(container, datapath, scope='all'))
    for item in items:
        params.append({key_name:item})
    return params

def on_start(container):

    parameters = collect_params(container, 'artifacts:*.cef.sourceAddress', 'ip')
    phantom.act('geolocate ip', parameters=parameters, name='my_geolocate_ip')

    return

def on_finish(container, summary):

    summary_json = phantom.get_summary()
    if 'result' in summary_json:
        for action_result in summary_json['result']:
            if 'action_run_id' in action_result:
                action_results = phantom.get_action_results(
                                    action_run_id=action_result['action_run _id'],
                                    result_data=False, flatten=False)
                phantom.debug(action_results)


    return
```

The return value of this API is a list of JSON dictionaries, a dictionary per app run which runs an instance for each asset that was used to run the action on that has the action_results parameter.

The following action_result JSON object is generated with the parameters result_data=False and flatten=False sent to the get_action_results() API in the playbook shown previously. If the parameter result_data was specified as True, the dictionaries in the action_results list must include data that has the full action result information. Setting the flatten parameter to True generates the same data but nested action_results data lists are reorganized to have a flat hierarchy with a list of higher level objects. This hierarchy is primarily for backward compatibility.

This sample uses the phantom.get_action_results() API.


```
[

    {
        "asset_id": 237,
        "status": "success",
        "name": "my_geolocate_ip",
        "app": "MaxMind",
        "action_results": [
            {
                "status": "success",
                "message": "Country: France",
```

```
                "parameter": {
                    "ip": "2.2.2.2",
                    "context": {...}
                },
                "summary": {
                    "country": "France"
                }
            },
            {
                "status": "success",
                "message": "Country: Australia",
                "parameter": {
                    "ip": "1.1.1.1",
                    "context": {...}
                },
                "summary": {
                    "country": "Australia"
                }
            }
        ],
        "app_id": 42,
        "app_run_id": 1076,
        "asset": "maxmind",
        "action": "geolocate ip",
        "message": "'my_geolocate_ip' on asset 'maxmind': 2 actions succeeded... ",
        "summary": { ...},
        "action_run_id": 1083
    }
]
```

get_apps

Use the get_apps API to let the user enumerate all of the apps installed on the system for each of the actions. The API returns a flat listing of all actions and apps with matching criteria.

The get_apps API is not supported from within a custom function.

```
phantom.get_apps(action, asset, app_type)
```

All of these parameters are optional, if the user does not specify any parameter, all the configured apps in the system are retrieved.

Parameter	Description
action	The name of the action. Use this parameter to retrieve information about assets that support the action. For example, name the action something like "block_ip" when you retrieve information about assets that support the block IP action.
asset	The asset name that allows users to retrieve only those apps that match the specified asset.
app_type	This allows users to retrieve only apps that match the specified type of the app.

This sample uses the phantom.get_apps() API.

```
def on_start(container):
    apps=[]
    apps = phantom.get_apps()
    phantom.debug(apps)
    apps=phantom.get_apps(action='file reputation')
    phantom.debug(apps)
    apps = phantom.get_apps(asset='my_smtp_asset')
    phantom.debug(apps)
```

```
    apps = phantom.get_apps(app_type='information')
    phantom.debug(apps)

    return
```

This sample shows the return value of this API, which is a list of JSON dictionaries that have the following schema.

```
[
  {
    "asset_disabled": false,
    "product_version_match": true,
    "app_type": "sandbox",
    "product_vendor": "Cuckoo",
    "product_name": "Cuckoo",
    "app_match_product_version": ".*",
    "asset_name": "cuckoo",
    "ap_name": "Cuckoo",
    "action": "detonate file",
    "app_version": "1.2.8",
    "asset_product_version": "",
    "asset_type": "sandbox"
  },
  ...

]
```

get_assets

Use the get_assets API if you have programmatic access to setup assets in the system.

The get_assets API is not supported from within a custom function.

```
phantom.get_assets(action=None, tags=None, types=None)
```

Parameter	Required	Description
action	Optional	The name of the action. Use this parameter to retrieve information about assets that support the action. For example, name the action something like "block_ip" when you retrieve information about assets that support the block IP action.
tags	Optional	A list of 'tags' used to retrieve assets that are tagged with the specified keyword.
types	Optional	A list of 'types' of assets that must be used to retrieve the specific assets.

This sample uses the phantom.get_assets() API.

```
def on_start(container):
    assets = phantom.get_assets()
    phantom.debug(assets)
    assets = phantom.get_assets(action='file reputation')
    phantom.debug(assets)
    assets = phantom.get_assets(types=['reputation service'])
    phantom.debug(assets)
    return
```

All of these parameters are optional, so if you don't specify any parameters, all the configured assets in the system are retrieved.

The sample shows the value of this API, which is a list of JSON dictionaries that have the following schema.

```
[
  {
    "description": "VirusTotal",
    "tags": [],
    "product_vendor": "VirusTotal",
    "product_version": "Private 2.0",
    "product_name": "VirusTotal",
    "disabled": true,
    "version": 1,
    "type": "reputation service",
    "id": 11,
    "name": "virustotal_private"
  },
  ...
]
```

get_container

Use the get_container API to retrieve the JSON for a container as a Python object.

The get_container API is supported from within a custom function.

```
json_object = phantom.get_container(container_id)
```

Parameter	Required?	Description
container_id	Required	The ID of the container.

This sample uses the phantom.get_container() API.

```
def on_start(container):

    cdata = phantom.get_container(container['id'])
    phantom.debug('Container Data: {}'.format(cdata))

    return
```

get_custom_function_results

Use the get_custom_function_results API to get the results of your custom function. This API returns the same structure that the callback function receives as a keyword argument.

The get_custom_function_results API is supported from within a custom function.

```
def get_custom_function_results(
    custom_function_run_id=None, custom_function_name=None, trace=False
):
```

Parameter	Required?	Description
custom_function_run_id	Optional. Provide either this parameter or the `custom_function_name` parameter.	The ID of the `CustomFunctionRun` object in the database. This is the value returned by the custom_function API.
custom_function_name	Optional. Provide either this parameter or the `custom_function_name` parameter.	The name of the custom function block. This name is unique for each playbook.

get_extra_data

Use the get_extra_data API to get the extra data retrieved during an action execution. You can specify the action, action run ID, or the app run ID as a key to obtain the data.

```
phantom.get_extra_data(action, action_run_id, app_run_id)
```

The get_extra_data API is not supported from within a custom function.

Parameter	Required?	Description
action	Optional	The action JSON object provided in the action callback. Using this parameter provides the extra data from the action that completed and triggered the callback function.
action_run_id	Optional	ID of the action run. Using this parameter obtains extra data from any completed action runs from the current playbook. The action run ID can be obtained from the previously noted action JSON object or by calling the phantom.get_summary() API, which enumerates all the actions that were executed in the playbook.
app_run_id	Optional	ID of the app run. Get the app run ID by calling the phantom.get_summary() API which enumerates all the actions that were executed in the playbook.

This sample uses the phantom.get_extra_data() API.

```
import phantom.rules as phantom
import json

def domain_reputation_cb(action, success, container, results, handle):
    if not success:
        return
    extra_data = phantom.get_extra_data(action)
    phantom.debug("Testing extra data: ")
    phantom.debug(extra_data)
    return


def on_start(container):
    phantom.act('domain reputation', parameters=[{ "domain" : "bjtuangouwang.com" }],
assets=["passivetotal"], callback=domain_reputation_cb)
    return

def on_finish(container, summary):
    phantom.debug("Summary: " + summary)
    return
```

The following sample shows the return value of this API, which is a list of JSON dictionaries that has the action results along with extra data.

```
[
  {
    "asset_id": 7,
    "extra_data": [
      {
        "status": "success",
        "extra_data": [{...}],
        "parameter": {}
      }
    ],
    "asset": "passivetotal"
  }
]
```

get_filtered_data

Use the get_filtered_data API to retrieve the filtered data that was saved by phantom.condition(). In the phantom.condition() API, if the name was specified, the filtered data is saved under the specified key and the same key can be used to retrieve the data. This API returns a tuple of filtered action results and filtered artifacts.

The get_filtered_data API is supported from within a custom function.

```
phantom.get_filtered_data(name=None)
```

Parameter	Required?	Description
name	Required	This parameter is used to save the filtered action results and filtered artifacts.

This sample uses the phantom.get_filtered_data() API.

```
import phantom.rules as phantom
import json
from datetime import datetime, timedelta

...

def filter_1(action=None,
             success=None,
             container=None,
             results=None,
             handle=None,
             filtered_artifacts=None,
             filtered_results=None):

    # collect filtered artifact ids for 'if' condition 1
    matched_artifacts_1, matched_results_1 = phantom.condition(
        container=container,
        action_results=results,
        conditions=[
            ["geolocate_ip_1:action_result.data.*.country_iso_code", "!=", "UK"],
            ["artifact:*.cef.bytesIn", "!=", 99],
        ],
        logical_operator='or',
        name="filter_1:condition_1")

...

def on_finish(container, summary):

    filtered_results, filtered_artifacts = phantom.get_filtered_data(name="filter_1:condition_1")
```

get_format_data

Use the get_format_data API to retrieve data saved through the phantom.format() API. If you specified the `name` parameter value in the phantom.format() API, the name can be used to retrieve the data. For sample usage, see format.

The get_format_data API is supported from within a custom function.

```
phantom.get_format_data(name=None)
```

get_parent_handle

Use the get_parent_handle API to retrieve the handle that has been set in the phantom.playbook() APIin the parent playbook. This API can be called from anywhere in the child playbook.

This API works only when the parent calls the child playbook in synchronous mode. See playbook for more information on calling playbooks in synchronous mode.

The get_parent_handle API is not supported from within a custom function.

```
json_object = phantom.get_parent_handle()
```

This sample shows the parent playbook used in the phantom.get_parent_handle() API.

```
        some_handle="some_handle from parent pb"
        # 'some_handle' is now passed to the child playbook through the handle parameter.
    playbook_run_id = phantom.playbook("local/child_pb", container=container,
name="playbook_local_child_pb_1", callback=decision_1, handle=some_handle)
```

This sample shows the child playbook used in the phantom.get_parent_handle() API.

```
def on_start(container):

    handle_from_parent=phantom.get_parent_handle() # this call can be done from any function of the child
playbook

    phantom.debug("handle sent by parent playbook: {}".format(handle_from_parent))

    return
```

get_playbook_info

Use the get_playbook_info API to retrieve your current playbook information such as ID, run ID, name, repository, and parent playbook run id, and the running playbook's effective user ID.

The get_playbook_info API is supported from within a custom function.

The return value of this API is a list containing a single dictionary.

```
phantom.get_playbook_info()
```

This sample uses the phantom.get_playbook_info() API.

```
[{
        'parent_playbook_run_id': '0',
        'name': 'test_plabook',
        'run_id': '37',
        'scope_artifacts': [],
        'scope': 'new',
        'id': '562',
        'repo_name': 'local',
    'effective_user_id':5
}]
```

get_raw_data

Use the get_raw_data API to retrieve container raw data as it exists at the source. This API allows users to access and automate on raw data in cases where there is information that was not parsed into artifacts.

The get_raw_data API is not supported from within a custom function.

```
phantom.get_raw_data(container)
```

Parameter	Required?	Description
container	Required	This is the JSON container object as available in `on_start`, callback, or `on_finish()` functions.

This sample uses the phantom.get_raw_data() API.

```
import phantom.rules as phantom
import json


def on_start(container):
    raw_data = phantom.get_raw_data(container)
    phantom.debug(raw_data)
    return

def on_finish(container, summary):
    return
```

The get_raw_data API pulls raw data from the container ["data"], and is often used to store raw emails and the ticketing tools raw data from `on_poll`. When pulling data, the API uses the ["data"] section of the container to do so.

This sample uses the phantom.get_raw_data() API.

```
phantom.debug(phantom.get_raw_data(container))
phantom.update(container, {"data": {"this": "is a test"}})
phantom.debug(phantom.get_raw_data(container))
in a custom block on a container that does not leverage container['data'].

The output:
Wed May 13 2020 11:08:38 GMT-0600 (Mountain Daylight Time): phantom.get_raw_data(): called for playbook run
'39792' and container id: '9420'
Wed May 13 2020 11:08:38 GMT-0600 (Mountain Daylight Time): {}
Wed May 13 2020 11:08:39 GMT-0600 (Mountain Daylight Time): successfully updated container(id: 9420)
Wed May 13 2020 11:08:39 GMT-0600 (Mountain Daylight Time): phantom.get_raw_data(): called for playbook run
'39792' and container id: '9420'
Wed May 13 2020 11:08:39 GMT-0600 (Mountain Daylight Time): {"this": "is a test"}
```

get_summary

Use the get_summary API to retrieve the summary of the playbook execution in a JSON format.

The get_summary API is not supported from within a custom function.

```
phantom.get_summary()
```

This sample uses the phantom.get_summary() API.

```
import phantom.rules as phantom
import json
```

```
def on_start(container):
    phantom.act('geolocate ip', parameters=[{ "ip" : "1.1.1.1" }])
    return

def on_finish(container, summary):
    summary_json = phantom.get_summary()
    phantom.debug(summary_json)
    return
```

This sample shows the return value of this API, which is a list of JSON representation of the playbook execution.


```
{
    "status": "success",
    "message": "",
    "result": [
        {
            "status": "success",
            "close_time": "2016-02-11T06:45:22.005343+00:00",
            "app_runs": [
                {
                    "asset_id": 40,
                    "status": "success",
                    "app": "MaxMind",
                    "app_id": 27,
                    "app_run_id": 224,
                    "asset": "maxmind",
                    "action": "geolocate ip",
                    "summary": "Country: Australia",
                    "parameter": "{\"ip\": \"1.1.1.1\"}",
                    "action_run_id": 104
                }
            ],
            "create_time": "2016-02-11T06:45:20.917+00:00",
            "action": "geolocate ip",
            "message": "1 action succeeded",
            "type": "investigate",
            "id": 104
        }
    ],
    "playbook_run_id": 167
}
```


parse_errors, print_errors, parse_success, parse_results

Use these APIs to pass in the action_results directly from callback into these helper routines to access the data. See collect before using this API, as these convenience APIs have limited use cases.

The parse_errors and parse_success APIs are supported from within a custom function.

```
phantom.parse_errors(action_results)
phantom.print_errors(action_results)
phantom.parse_success(action_results)
phantom.parse_results(action_results)
```

API	Description
parse_errors()	This API collects errors and returns the errors per asset and per parameter.
print_errors()	This API dumps any errors found in the `action_results` parameter.

API	Description
parse_success()	This API processes the `action_results` parameter and removes any records that had errors.
parse_results()	This API processes the `action_results` parameter and transforms the contents to be organized by success and failed categories.

set_parent_handle

Use the set_parent_handle API to set the handle from the synchronously called child playbook that is then accessed in the parent playbook through the handle parameter of the callback function. This API works only when the parent calls the child playbook in synchronous mode. See playbook for more information on calling playbooks in synchronous mode.

> The last call to the set_parent_handle API overwrites the handle sent to the callback function. In a parent playbook, if there is a join block where two child playbooks called synchronously are joining to a callback, the value of handle in the callback depends on which child playbook called the set_parent_handle API last.

The set_parent_handle API is not supported from within a custom function.

```
phantom.set_parent_handle()
```

The following sample uses the phantom.set_parent_handle() child playbook.

```
some_handle="some_handle from child pb"
phantom.set_parent_handle(some_handle)
```

The following sample uses the phantom.set_parent_handle() parent playbook.

```
def playbook_callback(..., handle=None, ...):

    phantom.debug("handle sent by child playbook: {}".format(handle))

    return
```

Session automation API

The Splunk SOAR (On-premises) Automation API allows security operations teams to develop detailed and precise automation strategies. Playbooks can serve many purposes, ranging from automating minimal investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of session automation using playbooks.

build_phantom_rest_url

Use the build_phantom_rest_url API to combine the Splunk SOAR (On-premises) base URL and the specific resource path, such as `/rest/artifact`.

The build_phantom_rest_url API is supported from within a custom function.

```
phantom.build_phantom_rest_url()
```

This sample uses the phantom.build_phantom_rest_url API.

```
build_phantom_rest_urll(*args)
    Constructs a REST route given a list of part paths. Quotes each path part individually.
```

```
    Note that you must not pass in quoted endpoints, or they'll get re-quoted.

    Examples:
    >>> build_phantom_rest_url()
    'https://127.0.0.1/rest/'

    >>> build_phantom_rest_url('container')
    'https://127.0.0.1/rest/container'

    >>> build_phantom_rest_url('container', 7)
    'https://127.0.0.1/rest/container/7'

    >>> build_phantom_rest_url('container', '7/')
    'https://127.0.0.1/rest/container/7'

    >>> build_phantom_rest_url('container/7/edit_options')
    'https://127.0.0.1/rest/container/7/edit_options'

    >>> build_phantom_rest_url('decided_list', 'foo bar')
    'https://127.0.0.1/rest/decided_list/foo%20bar'

    Args:
        *args (str|int): args to join together

    Returns:
        str
```

get_base_url

Use the get_base_url API to retrieve the URL that points to your Splunk SOAR (On-premises) instance.

The get_base_url API is supported from within a custom function.

```
phantom.get_base_url()
```

This sample uses the phantom.get_base_url() API.

```
import phantom.rules as phantom
import json

def on_start(container):
    url = phantom.get_base_url()
    phantom.debug(url)
    return

def on_finish(container, summary):
    return
```

The return value is the base URL of the platform as configured in **Administration > System Settings > Company Settings**.

```
2016-02-13T01:29:25.977000+00:00: calling on_start(): on incident 'test', id: 107.
2016-02-13T01:29:26.020179+00:00: phantom.get_base_url(): called for playbook run '219'
2016-02-13T01:29:26.022549+00:00: https://10.10.0.10
2016-02-13T01:29:26.025000+00:00: No actions were executed
2016-02-13T01:29:26.034220+00:00: calling on_finish()
2016-02-13T01:29:26.049943+00:00:
Playbook 'get_base_url (id: 175)' executed (playbook_run_id: 219) on incident 'test'(id: 107).
Playbook execution status is:'success'
        No actions were executed for this playbook and 'incident'
```

```
{"message":"No actions were executed","playbook_run_id":219,"result":[],"status":"success"}

 *** The Playbook has completed. Result: success ***
```

get_phantom_home

Use the get_phantom_home API to return the path to the Splunk SOAR (On-premises) home directory.

The get_phantom_home is supported from within a custom function.

```
phantom.get_phantom_home()
```

The following examples show the return values for the get_phantom_home API on different types of installs.

On a privileged install:

```
Thu Jan 03 2019 16:37:23 GMT-0800 (Pacific Standard Time): /opt/phantom
```

On an unprivileged install:

```
Thu Jan 03 2019 16:36:31 GMT-0800 (Pacific Standard Time): /home/username/directory_name
```

get_rest_base_url

Use the get_rest_base_url API to return the base URL to the REST API of your Splunk SOAR (On-premises) instance. This API works on all Splunk SOAR (On-premises) instances, regardless of installation type, or the base URL found in the in **Company Settings**.

The get_rest_base_url is supported from within a custom function.

```
phantom.get_rest_base_url()
```

The following examples show the return values for the get_rest_base_url API on different types of installs.

On a privileged install:

```
Thu Jan 03 2019 16:37:23 GMT-0800 (Pacific Standard Time): https://127.0.0.1/rest/
```

On an unprivileged install, showing the custom HTTPS port:

```
Thu Jan 03 2019 16:36:31 GMT-0800 (Pacific Standard Time): https://127.0.0.1:9999/rest/
```

requests

Use the requests API to interact with the Splunk SOAR (On-premises) platform through the REST API. By using phantom.requests instead of directly importing requests from site packages, you avoid the need to set the Splunk SOAR (On-premises) request headers by hand, meaning that you don't need to authenticate with the platform. For more information on how to use the requests package, see https://pypi.org/project/requests.

The requests API is supported from within a custom function.

```
phantom.requests
```

This sample uses the phantom.requests() API.

```
def list_indicator_tags(indicator_id=None, **kwargs):
    """
    List the tags on the indicator with the given ID

    Args:
        indicator_id: The ID of the indicator to list the tags for

    Returns a JSON-serializable object that implements the configured data paths:
        tags: The tags associated with the given indicator
    """
    ############################ Custom Code Goes Below This Line #################################
    import json
    import phantom.rules as phantom
    outputs = {}

    # Validate the input
    if indicator_id is None:
        raise ValueError('indicator_id is a required parameter')

    # phantom.build_phantom_rest_url will join positional arguments like you'd expect (with URL encoding)
    indicator_tag_url = phantom.build_phantom_rest_url('indicator', indicator_id, 'tags')

    # Using phantom.requests ensures the correct headers for authentication
    response = phantom.requests.get(
        indicator_tag_url,
        verify=False,
    )
    phantom.debug("phantom returned status code {} with message {}".format(response.status_code,
response.text))

    # Get the tags from the HTTP response
    indicator_tag_list = response.json()['tags']
    phantom.debug("the following tags were found on the indicator: {}".format(indicator_tag_list))
    outputs['tags'] = indicator_tag_list

    # Return a JSON-serializable object
    assert json.dumps(outputs)  # Will raise an exception if the :outputs: object is not JSON-serializable
    return outputs
```

set_action_limit

Use `set_action_limit` in your playbook's `on_start()` block to set the maximum number of action calls that can be executed. The default is 50 action calls per container per Playbook. Each phantom.act() call can still result in multiple actions performed, resulting in more actions than this setting.

`set_action_limit` is not supported from within a custom function.

```
phantom.set_action_limit(limit)
```

Vault automation API

The Splunk SOAR (On-premises) Automation API allows security operations teams to develop detailed and precise automation strategies. Playbooks can serve many purposes, ranging from automating minimal investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of vault automation using playbooks.

Vault background

The Vault is used to store container attachments, also known as artifacts. Users can access the vault using the vault playbook API, described here, or by directly uploading attachments to a container. The vault accepts files of all types and sizes. Vault files are stored on your drive under `PHANTOM_HOME/vault/`. Files are renamed using sha1 hashes and are saved to the database.

Vault files are duplicated to all SOAR instances in a clustered environment. When a container is deleted, all container attachments are removed and all vault files associated with the attachments are deleted from your `PHANTOM_HOME/vault/` directory. In a warm standby situation, the vault directory is synced from the primary instance to the warm backup through Rsync. See Rsync in the *Administer Splunk SOAR (On-premises)* manual. To access the vault through the Splunk SOAR (On-premises) visual playbook editor, use custom code. For details on custom code, see Add custom code to your Splunk SOAR (On-premises) playbook with the code block in the Splunk SOAR (On-premises) *Build Playbooks with the Playbook Editor* manual.

vault_add

Use the vault_add API to attach a file to a container by adding it to the vault. This API returns a success flag, any response message, and the vault ID of the vault file.

The vault_add API is supported from within a custom function.

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

Parameter	Required?	Type	Description
container	Optional	Integer or dictionary	The container to act on. This parameter can either be a numerical ID or a container object. If no container is provided, the currently running container is used.
file_location	Required	String	This parameter is the location of the file on the Splunk SOAR (On-premises) file system. Write this file to `/opt/phantom/vault/tmp/` directory before calling this API.
file_name	Optional	String	A custom file name. This parameter shows up as the file name in the container's vault files list.
metadata	Optional	Dictionary	Metadata about the file. Currently the only user-supplied attribute that is used is "contains", which specifies what kind of information is available. For example, the `metadata={"contains":["pe file"]}` syntax tells the system the file is a Windows PE file, which enables actions such as "detonate file".
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

The following is an example code snippet that can be used to add a file to the vault from the playbook:

phantom.vault_add() API.

```
import phantom.rules as phantom
import json


def on_start(container):

    return

def on_finish(container, summary):

    phantom.debug('phantom.vault_add start')
```

```
    success, message, vault_id = phantom.vault_add(
        file_location="/opt/phantom/vault/tmp/myfile",
        file_name="notepad.exe",
        metadata={"contains": ["PE File"]}
    )

    phantom.debug(
        'phantom.vault_add results: success: {}, message: {}, vault_id: {}'\
        .format(success, message, vault_id)
    )

    return
```

vault_delete

Use the vault_delete API to remove a file from the vault. This API returns a tuple of a success flag, any response message, and the list of deleted file names.

The vault_delete API is supported from within a custom function.

```
phantom.vault_delete(vault_id=None, file_name=None, container_id=None, remove_all=False, trace=False)
```

Parameter	Required?	Type	Description
vault_id	Optional, unless the `file_name` is not provided	String	The alphanumeric file hash of the vault file, such as `41c4e1e9abe08b218f5ea60d8ae41a5f523e7534`.
file_name	Optional, unless the `vault_id` is not provided.	String	Name of the file to delete.
container_id	Optional	Integer	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()`.
remove_all	Optional	Boolean	If multiple items are found with the same identifier, set this parameter to True to remove all. If multiple files are found and this parameter is set to False, an error is returned.
trace	Optional	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

For this API, you need to give the automation user permissions to delete containers.

This sample uses the vault_delete() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.vault_delete start')

    success, message, deleted_files = phantom.vault_delete(
        vault_id='41c4e1e9abe08b218f5ea60d8ae41a5f523e7534',
        remove_all=False
    )

    phantom.debug(
        'phantom.vault_delete results: success: {}, message: {}, deleted_files: {}'\
        .format(success, message, deleted_files)
```

```
    )

    return
```

vault_info

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

The vault_info API is supported from within a custom function.

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

Parameter	Required?	Type	Description
vault_id	Optional	String	The alphanumeric file hash of the vault file, such as `41c4e1e9abe08b218f5ea60d8ae41a5f523e7534`.
file_name	Optional	String	The file name of the vault file.
container_id	Optional	Integer	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	Boolean	Trace is a flag related toÂ the level of logging. If trace is on (True), more logging is enabled. When set to True, more detailed output is displayed in debug output.

Example request

This sample uses the phantom.vault_info() API.

```
import phantom.rules as phantom

def on_start(container):

    phantom.debug('phantom.vault_info start')

    success, message, info = phantom.vault_info(
        vault_id='41c4e1e9abe08b218f5ea60d8ae41a5f523e7534',
        container_id=1
    )

    phantom.debug(
        'phantom.vault_info results: success: {}, message: {}, info: {}'\
        .format(success, message, info)
    )

    return
```

Example response

The following is an example of what the API returns within the third tuple member.

```
[
    {
        "container": "Incident # 1",
        "name": "ping.exe",
        "aka": [
            "ping.exe"
        ],
        "metadata": {
```

```
            "contains": [
                "pe file"
            ],
            "sha256": "14262982a64551fde126339b22b993b6e4aed520e53dd882e67d887b6b66f942",
            "md5": "5fb30fe90736c7fc77de637021b1ce7c",
            "sha1": "41c4e1e9abe08b218f5ea60d8ae41a5f523e7534"
        },
        "id": 3,
        "container_id": 1,
        "create_time": "37 minutes ago",
        "vault_id": "41c4e1e9abe08b218f5ea60d8ae41a5f523e7534",
        "user": "",
        "vault_document": 3,
        "hash": "41c4e1e9abe08b218f5ea60d8ae41a5f523e7534",
        "path": "/opt/phantom/vault/41/c4/41c4e1e9abe08b218f5ea60d8ae41a5f523e7534",
        "size": 16896
    },
....
....
]
```

Network automation API

The Splunk SOAR (On-premises) Automation API allows security operations teams to develop detailed and precise automation strategies. Playbooks can serve many purposes, ranging from automating minimal investigative tasks that can speed up analysis to large-scale responses to a security breach. The following APIs are supported to leverage the capabilities of network automation using playbooks.

address_in_network

The address_in_network API checks if the IP address is in the user-specified IP address range is expressed in CIDR format.

The address_in_networkis API is supported from within a custom function.

```
phantom.address_in_network(ip, net)
```

Parameter	Description
ip	This parameter is the IPv4 address that has to be checked.
net	This parameter is the IPv4 CIDR notation expressing the IP address range that needs to be tested.

This sample uses the phantom.address_in_network() API.

```
phantom.address_in_network('192.168.100.11', '192.168.100.0/24')
```

The address must be within the address range to return true. For example, phantom.valid_ip('phantom.address_in_network('192.168.100.11', '10.0.0.0/8')') returns false.

attacker_ips, victim_ips

Review collect before using either of these APIs, as these convenience APIs have limited use cases. These APIs return an attacker or victim value depending on the CEF `deviceDirection`, `sourceAddress`, and `destinationAddress` fields.

The attacker_ips and victim_ips APIs are supported from within a custom function.

```
phantom.attacker_ips(container, scope='new')
phantom.victim_ips(container, scope='new')
```

Parameter	Description
container	This is the container object passed in to the on_start() API or any action callbacks.
scope	For more details about this parameter see collect. The parameter defaults to 'new' or you can pass 'all' to collect the field values from all artifacts.

- If the deviceDirection field is inbound or not present, the `sourceAddress` field is returned as the attacker IP address and `destinationAddress` is returned as the victim IP address.
- If the `deviceDirection` field is outbound, then the `destinationAddress` field is returned as the attacker IP address and `sourceAddress` is returned as the victim IP address.

valid_ip

The valid_IP API validates an IPv4 address.

```
phantom.valid_ip(address)
```

Parameter	Required?	Description
address	Required	This parameter validates the IPv4 address

This sample uses the phantom.valid_ip() API.

```
phantom.valid_ip('192.168.100.11')
```

The IPv4 address format must be used to return true. The host name, URL, or domain, such as phantom.valid_ip('https://my.phantom.us'), returns false.

valid_net

The valid_net API validates a CIDR notation of IPv4 address range.

```
phantom.valid_net(net)
```

Parameter	Required?	Description
net	Required	This parameter validates the CIDR notation of IPv4 address range, such as /0 or /32.

This sample uses the phantom.valid_net() API.

```
phantom.valid_net('192.168.100.11/32')
```

The CIDR notation format must be used to return true. The hostname, URL, or domain, such as phantom.valid_ip('https://my.phantom.us'), returns false.