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

Generated: 10/27/2022 8:33 pm

Table of Contents

Playbook API Tutorial

Python Playbook Tutorial for Splunk SOAR (On-premises) overview

Playbooks help security operations teams develop and deploy precise automation strategies. These strategies might range from generic information mining tasks to actively mitigating the impact of an ongoing incident.

Playbooks are Python scripts that execute various actions in response to an incident. As new data enters the system, enabled playbooks run on new containers in a specified order. Configure playbooks to act on containers with a specific label. For example, if the imported data has a label of `incident`, the playbook is expected to run on an incident. All playbooks are independent of each other and can act on all containers. See Understanding containers in the *Python Playbook API Reference for Splunk SOAR (On-premises)* for more information on containers.

Playbooks have the following states:

Playbook state	Description
Active	Execute automatically when a new artifacts is added to an unresolved container. A user can also run any playbook manually.
Safe Mode	Only execute read-only actions. For example, `list vms` is a read-only action, whereas `block ip`, which could result in changes to the state of the asset, isn't.

Active playbooks run in real-time as new containers are imported into the system or existing open containers are updated with new artifacts. Playbooks don't automatically run if a container is in the closed state. However, new artifacts can continue to be added to the container and users can still run actions or playbooks on them manually.

Playbook structure

Playbooks are Python scripts built to run on top of the playbook API platform. Every playbook has two special functions called `on_start()` and `on_finish()`, which are called by the platform at the beginning and end of the playbook execution. Expand the following section to see a minimal playbook consisting of just one action:

Phantom playbook functions

```
import phantom.rules as phantom
import json

# This function gets called for all new containers or when new artifacts
# are added to an existing container.
def on_start(container):

    # container is a JSON object representing the object that this playbook
    # can automate on

    # use phantom.collect() API to get the artifacts that belong
    # to this container and call phantom.act().

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

```
    if parameters:
        # In phantom.act(), use the optional parameter 'callback' to get
        # called when the action completes
        my_test_ips = []
        my_test_ips.append('1.1.1.1')
        phantom.act('geolocate ip',
                    parameters=parameters,
                    assets=["maxmind"],
                    handle=json.dumps(my_test_ips),
                    callback=geolocate_ip_cb)
    return

# this is a callback function for the action.
# Evaluate the results and/or call other actions
def geolocate_ip_cb(action, success, container, results, handle):

    #
    # See the documentation for 'callback' for detailed information on
    # these parameters
    #

    if not success:
        return

    return

def on_finish(container, summary):
    # Summary is a user friendly representation of all action outcomes

    # call phantom.get_summary() to get a JSON representation of all
    # action results here
    return
```

Don't use globals in Splunk SOAR (On-premises) playbooks as their state isn't preserved in some cases. Instead, use `handle`, a parameter to the phantom.act() API, to pass objects between an action and its callbacks. Alternatively, you can use the phantom.save_data and phantom.get_data APIs to save and retrieve objects.

Function	Description

`on_start` (container)	Every time a container is ingested or updated, Splunk SOAR (On-premises) calls the `on_start()` function of all enabled playbooks. To get the artifacts of a container on which to act, a user can call the phantom.collect() API to extract the artifacts.
	
	Example 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",`

Function	Description

	<pre> "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
	}</pre>

`on_finish` (container, summary)	The final method invoked after all actions run for a given container. The parameter summary presents the final outcome of the playbook execution in a human readable format. A user can call the phantom.get_summary() API to get a JSON representation of the summary of the playbook execution.

Custom lists

Custom lists are data structures that store information that you can access through playbooks for real-time complex decision making. For example, a playbook might take different actions if an IP address is a critical server versus a test machine. To do so, it must check for the attacked IP address and its list membership.

Users can configure custom lists of test machine IP addresses, finance department endpoint IP addresses, and executives by first and last name or email address. Playbooks can then directly reference these custom lists.

Playbook versioning

Each time a user modifies and saves a playbook, its version is automatically incremented. Automation only uses the latest version of each playbook.

If a user edits a rule while a playbook is running, Splunk SOAR (On-premises) continues to execute the playbook in the state in which the execution started. Any changes to the later version of the playbook only apply to a new execution instance for a new incident, or an incident with new information.

Common API calls used by the Visual Playbook Editor

The Visual Playbook Editor uses the `phantom.act` API calls to perform actions in a playbook. The `phantom.collect` API calls are used to collect input for actions.

Run actions using the phantom.act API calls

`phantom.act` is the call for running an action. The following code shows an example of a simple act call. In this act call, the parameters are static as shown by the 1.1.1.1 ip. This means that regardless of which container you run against, the playbook will work on the same input data. In the following act call, there is no callback.

```
phantom.act(action="geolocate ip", parameters=parameters, assets=['maxmind'], name="geolocate_ip_1")
```

It is important to note that in the call to phantom.act, the callback parameter isn't set, and you can see that in the logs for the playbook run and the log output.

View the simplest phantom.act call.

```
"""
"""

import phantom.rules as phantom
import json
from datetime import datetime, timedelta
def on_start(container):
    phantom.debug('on_start() called')

    # call 'geolocate_ip_1' block
    geolocate_ip_1(container=container)

    return

def geolocate_ip_1(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None, custom_function=None):
    phantom.debug('geolocate_ip_1() called')

    # collect data for 'geolocate_ip_1' call

    parameters = []

    # build parameters list for 'geolocate_ip_1' call
    parameters.append({
        'ip': "1.1.1.1",
    })

    phantom.act(action="geolocate ip", parameters=parameters, assets=['maxmind'], name="geolocate_ip_1")

    return

def on_finish(container, summary):
    phantom.debug('on_finish() called')
    # This function is called after all actions are completed.
    # summary of all the action and/or all details of actions
    # can be collected here.

    # 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
```

Using the callback parameter

Use the callback function to chain playbook blocks together that rely on data from the previous blocks. If you'd like a later playbook block to rely on the output of your act call, use a callback function. In the following example, the filter block relies on the output of the geolocate ip command.

View a phantom.act call that uses a callback.

```
"""
```

```
"""

import phantom.rules as phantom
import json
from datetime import datetime, timedelta
def on_start(container):
    phantom.debug('on_start() called')

    # call 'geolocate_ip_1' block
    geolocate_ip_1(container=container)

    return

def geolocate_ip_1(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None, custom_function=None):
    phantom.debug('geolocate_ip_1() called')

    # collect data for 'geolocate_ip_1' call

    parameters = []

    # build parameters list for 'geolocate_ip_1' call
    parameters.append({
        'ip': "1.1.1.1",
    })

    phantom.act(action="geolocate ip", parameters=parameters, assets=['maxmind'], callback=filter_1,
name="geolocate_ip_1")

    return

def filter_1(action=None, success=None, container=None, results=None, handle=None, filtered_artifacts=None,
filtered_results=None, custom_function=None):
    phantom.debug('filter_1() called')

    # 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", "==", "AU"],
        ],
        name="filter_1:condition_1")

    # call connected blocks if filtered artifacts or results
    if matched_artifacts_1 or matched_results_1:
        lookup_ip_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)

    return

def lookup_ip_1(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None, custom_function=None):
    phantom.debug('lookup_ip_1() called')

    #phantom.debug('Action: {0} {1}'.format(action['name'], ('SUCCEEDED' if success else 'FAILED')))

    # collect data for 'lookup_ip_1' call
    filtered_results_data_1 = phantom.collect2(container=container,
datapath=["filtered-data:filter_1:condition_1:geolocate_ip_1:action_result.parameter.ip",
"filtered-data:filter_1:condition_1:geolocate_ip_1:action_result.parameter.context.artifact_id"])
```

```
    parameters = []

    # build parameters list for 'lookup_ip_1' call
    for filtered_results_item_1 in filtered_results_data_1:
        if filtered_results_item_1[0]:
            parameters.append({
                'ip': filtered_results_item_1[0],
                # context (artifact id) is added to associate results with the artifact
                'context': {'artifact_id': filtered_results_item_1[1]},
            })

    phantom.act(action="lookup ip", parameters=parameters, assets=['google_dns'], name="lookup_ip_1")

    return

def on_finish(container, summary):
    phantom.debug('on_finish() called')
    # This function is called after all actions are completed.
    # summary of all the action and/or all details of actions
    # can be collected here.

    # 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
```

To learn more about the callback function, see callback in the *Python Playbook API Reference for Splunk SOAR (On-premises)*.

Collect input using phantom.collect API calls

Collect APIs are used to gather data from artifacts, containers, and action results. The data is then used in filtering, or used as input to act calls. The input to this phantom.act call is determined by the container data. In the following example, the geolocate ip action is using the input container artifacts for its data input.

Phantom.act calls also often use collect2 output. With collect2, the generated code loops through the list of results from collect2, and then appends a dictionary to the list parameters, with each row containing the ip and context parameters. This parameter list is passed to the phantom.act call.

```
def geolocate_ip_1(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None):
    phantom.debug('geolocate_ip_1() called')

    # collect data for 'geolocate_ip_1' call
    container_data = phantom.collect2(container=container, datapath=['artifact:*.cef.sourceAddress',
'artifact:*.id'])

    parameters = []

    # build parameters list for 'geolocate_ip_1' call
    for container_item in container_data:
        if container_item[0]:
            parameters.append({
```

```
                'ip': container_item[0],
                # context (artifact id) is added to associate results with the artifact
                'context': {'artifact_id': container_item[1]},
            })

    phantom.act("geolocate ip", parameters=parameters, assets=['maxmind'], name="geolocate_ip_1")

    return
```

The purpose of the collect call is to extract the requested set of data from the container. In this example, the collect call requested the sourceAddress CEF artifacts. To get that set of artifacts, we use a datapath with the collect call.

Using phantom.act calls with collect2 output

The collect2 call can pre-filter results. The Visual Playbook Editor (VPE) generates code of this type when you use a decision block. This is shown in this filter block called by on_start:

```
def filter_1(action=None, success=None, container=None, results=None, handle=None, filtered_artifacts=None,
filtered_results=None):
    phantom.debug('filter_1() called')

    # collect filtered artifact ids for 'if' condition 1
    matched_artifacts_1, matched_results_1 = phantom.condition(
        container=container,
        conditions=[
            ["artifact:*.cef.sourceAddress", "<", "10.0.0.0"],
            ["artifact:*.cef.sourceAddress", ">", "10.255.255.255"],
        ],
        logical_operator='or',
        name="filter_1:condition_1")

    # call connected blocks if filtered artifacts or results
    if matched_artifacts_1 or matched_results_1:
        geolocate_ip_1(action=action, success=success, container=container, results=results, handle=handle,
filtered_artifacts=matched_artifacts_1, filtered_results=matched_results_1)

    return
```

This is passed to the geolocate block as a parameter, and then geolocate block uses it in the phantom.collect2 call as shown here:

```
def geolocate_ip_1(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None):
    phantom.debug('geolocate_ip_1() called')

    # collect data for 'geolocate_ip_1' call
    container_data = phantom.collect2(container=container, datapath=['artifact:*.cef.sourceAddress',
'artifact:*.id'])

    parameters = []

    # build parameters list for 'geolocate_ip_1' call
    for container_item in container_data:
        if container_item[0]:
            parameters.append({
                'ip': container_item[0],
                # context (artifact id) is added to associate results with the artifact
                'context': {'artifact_id': container_item[1]},
            })
```

```
    phantom.act("geolocate ip", parameters=parameters, assets=['maxmind'], name="geolocate_ip_1")

    return
```

The datapath for the geolocate ip command starts with "filtered-artifacts:", so you are using the results of the filtered set, instead of all artifacts.

For more information on the phantom.act and phantom.collect API calls, see *Splunk SOAR (On-premises) API Reference*.

phantom.debug call

If you aren't getting the results you expected with your API calls, add a phantom.debug call below the collect2 call, as shown in the following example:

```
container_data = phantom.collect2(container=container, datapath=['artifact:*.cef.sourceAddress',
'artifact:*.id'])
phantom.debug(container_data)
```

Once the collect2 call is debugged, the additional output in the debug window shows the sourceAddress artifacts and their artifact IDs.

```
Wed Mar 21 2018 20:32:37 GMT-0700 (PDT):
[
    [
        "10.10.10.10",
        2
    ],
    [
        "175.45.176.1",
        3
    ]
]
```

`phantom.debug` prints the Python representation of the object that you pass to it.

```
container_data = phantom.collect2(container=container, datapath=['artifact:*.cef', 'artifact:*.id'])
phantom.debug(container_data)
```

The debug output is this:

```
Wed Mar 21 2018 20:35:43 GMT-0700 (PDT):
[
    [
        {
            "destinationAddress": "192.168.10.10",
            "sourceAddress": "10.10.10.10"
        },
        2
    ],
    [
        {
            "destinationAddress": "192.168.10.11",
            "sourceAddress": "175.45.176.1"
        },
        3
    ]
]
```

Tutorial: Create a simple playbook in Splunk SOAR (On-premises)

One of the simplest examples of a playbook is one that executes a single action and does nothing with the results. The following `geolocate ip` action is one such example:

```
import phantom.rules as phantom

def on_start(container):

    phantom.act(action="geolocate ip",
                parameters=[{'ip': "1.1.1.1"}])

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

The only parameter required by the `geolocate ip` action is provided with a hard-coded value. Because you haven't specified an asset or an asset tag to operate on, Splunk SOAR (On-premises) finds all assets that have a supporting app which support the `geolocate ip` action and runs the action against each of them. In a default Splunk SOAR (On-premises) installation, this will only use the MaxMind app, but other possibilities include services like FreeGeoIP and HackerTarget.

Incorporate a callback

The previous example executed the first action. While it queried MaxMind for the location of an IP address, it didn't leverage the results or take any further action. Because Splunk SOAR (On-premises) runs all actions asynchronously to minimize waiting periods, you must incorporate a callback function to parse the results and act on the location information discovered by the `geolocate ip` action.

To incorporate a callback, pass the `callback` parameter to the `act()` function with a callback function that receives these results. The following example shows how to incorporate a callback:

Example

Example function

```
import phantom.rules as phantom

def on_start(container):

    phantom.act(action="geolocate ip",
                parameters=[{'ip': "1.1.1.1"}],
                callback=geolocate_ip_cb
               )
def geolocate_ip_cb(action=None, success=None, container=None, results=None, handle=None,
filtered_artifacts=None, filtered_results=None, custom_function=None):

    if not success:
        return

    return

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

For more information about the `callback` function, see callback in the *Python Playbook API Reference for Splunk SOAR (On-premises)*.

Insert debug statements

Add the `debug()` function to insert debug statements that you can see when running your playbook on a container. The following example demonstrates how to insert debug statements:

Example

Example function

```
import phantom.rules as phantom

def on_start(container):

    phantom.act(action="geolocate ip",
                parameters=[{'ip': "1.1.1.1"}],
                callback=geolocate_ip_cb
               )

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

    if not success:
        phantom.debug("failed action: {}".format(action))
        return

    phantom.debug("successful action: {}".format(action))
    return

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

The function produces debug output in the debug console when you execute your playbook. Use the output to inspect and debug your playbook before deploying it in production.

Example output

```
Fri May 01 2020 11:42:36 GMT-0400 (Eastern Daylight Time): Starting playbook 'local/tutorial (version: 10,
id: 76)' on 'events'(id: 1) with playbook run id: 10, running as user 'admin(root@localhost)'(id: 1) with
scope 'all'
Fri May 01 2020 11:42:36 GMT-0400 (Eastern Daylight Time): calling on_start() on events 'tester 1'(id: 1).
Fri May 01 2020 11:42:36 GMT-0400 (Eastern Daylight Time): phantom.act(): for action 'geolocate ip' no
assets were specified and hence the action shall execute on all assets the app (supporting the action) can
be executed on
Fri May 01 2020 11:42:36 GMT-0400 (Eastern Daylight Time): phantom.act(): action 'geolocate ip' on assets:
maxmind, callback function: 'geolocate_ip_cb', with no action reviewer, no delay to execute the action, no
user provided name for the action, no tags, no asset type
Fri May 01 2020 11:42:36 GMT-0400 (Eastern Daylight Time): 'geolocate ip' shall be executed now on asset
'maxmind'(id:2) using app 'MaxMind'
Fri May 01 2020 11:42:36 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': app 'MaxMind'
started successfully. Execution parameters sent.
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': Loaded action
execution configuration
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': MaxMind DB
loaded
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': 1 action
succeeded. (1)For Parameter: {"ip":"1.1.1.1"} Message: "City: Research, State: VIC, Country: Australia"
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): 'geolocate ip'  on asset 'maxmind' completed with
status: 'success'. Action Info: [{"app_name":"MaxMind","asset_name":"maxmind","param":{"ip": "1.1.1.1",
```

```
"context": {"guid": "945257ce-b1f4-424a-9541-f11fd8bb48bd", "artifact_id": 0, "parent_action_run":
[]}},"status":"success","message":"City: Research, State: VIC, Country: Australia"}]
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): calling action 'geolocate ip''s callback function
'geolocate_ip_cb()'
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): successful action: {'action': 'geolocate ip',
'action_name': 'geolocate ip', 'type': 'investigate', 'action_run_id': 9, 'name': 'geolocate ip'}
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): finished action 'geolocate ip''s callback
function 'geolocate_ip_cb()'
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time):

Playbook 'tutorial' (playbook id: 76) executed (playbook run id: 10) on events 'tester 1'(container id: 1).
   Playbook execution status is 'success'
        Total actions executed: 1
        Action 'geolocate ip'(geolocate ip)
           Status: success
                App 'MaxMind' executed the action on asset 'maxmind'
                   Status: success
                   Parameter: {"ip":"1.1.1.1"}

Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): *** Playbook 'local/tutorial' execution (10) has
completed with status: SUCCESS ***
Fri May 01 2020 11:42:37 GMT-0400 (Eastern Daylight Time): 1 action succeeded
```

Perform final actions

After the playbook runs and all actions finish executing, call the `on_finish()` handler to perform any final actions. For example, you might send a summary email, close an incident, or update a ticketing system with a status.

The following example shows how to incorporate an `on_finish()` handler:

Example

Example function

```
import phantom.rules as phantom

def on_start(container):

    phantom.act(action="geolocate ip",
                parameters=[{'ip': "1.1.1.1"}],
                callback=geolocate_ip_cb
               )

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

    if not success:
        phantom.debug("failed action: {}".format(action))
        return

    phantom.debug("successful action: {}".format(action))
    return

def on_finish(container, summary):

    phantom.debug(summary)
    return
```

Example response

```
Fri May 01 2020 11:44:14 GMT-0400 (Eastern Daylight Time): Starting playbook 'local/tutorial (version: 11,
id: 77)' on 'events'(id: 1) with playbook run id: 11, running as user 'admin(root@localhost)'(id: 1) with
scope 'all'
Fri May 01 2020 11:44:14 GMT-0400 (Eastern Daylight Time): calling on_start() on events 'tester 1'(id: 1).
Fri May 01 2020 11:44:14 GMT-0400 (Eastern Daylight Time): phantom.act(): for action 'geolocate ip' no
assets were specified and hence the action shall execute on all assets the app (supporting the action) can
be executed on
Fri May 01 2020 11:44:14 GMT-0400 (Eastern Daylight Time): phantom.act(): action 'geolocate ip' on assets:
maxmind, callback function: 'geolocate_ip_cb', with no action reviewer, no delay to execute the action, no
user provided name for the action, no tags, no asset type
Fri May 01 2020 11:44:14 GMT-0400 (Eastern Daylight Time): 'geolocate ip' shall be executed now on asset
'maxmind'(id:2) using app 'MaxMind'
Fri May 01 2020 11:44:14 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': app 'MaxMind'
started successfully. Execution parameters sent.
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': Loaded action
execution configuration
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': MaxMind DB
loaded
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): 'geolocate ip' on asset 'maxmind': 1 action
succeeded. (1)For Parameter: {"ip":"1.1.1.1"} Message: "City: Research, State: VIC, Country: Australia"
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): 'geolocate ip'  on asset 'maxmind' completed with
status: 'success'. Action Info: [{"app_name":"MaxMind","asset_name":"maxmind","param":{"ip": "1.1.1.1",
"context": {"guid": "16cc5fdc-88d1-42b3-9385-fdcdf3700bd2", "artifact_id": 0, "parent_action_run":
[]}},"status":"success","message":"City: Research, State: VIC, Country: Australia"}]
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): calling action 'geolocate ip''s callback function
'geolocate_ip_cb()'
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): successful action: {'action': 'geolocate ip',
'action_name': 'geolocate ip', 'type': 'investigate', 'action_run_id': 10, 'name': 'geolocate ip'}
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): finished action 'geolocate ip''s callback
function 'geolocate_ip_cb()'
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time):

Playbook 'tutorial' (playbook id: 77) executed (playbook run id: 11) on events 'tester 1'(container id: 1).
   Playbook execution status is 'success'
        Total actions executed: 1
        Action 'geolocate ip'(geolocate ip)
           Status: success
                App 'MaxMind' executed the action on asset 'maxmind'
                   Status: success
                   Parameter: {"ip":"1.1.1.1"}

Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time):

Playbook 'tutorial' (playbook id: 77) executed (playbook run id: 11) on events 'tester 1'(container id: 1).
   Playbook execution status is 'success'
        Total actions executed: 1
        Action 'geolocate ip'(geolocate ip)
           Status: success
                App 'MaxMind' executed the action on asset 'maxmind'
                   Status: success
                   Parameter: {"ip":"1.1.1.1"}

Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): *** Playbook 'local/tutorial' execution (11) has
completed with status: SUCCESS ***
Fri May 01 2020 11:44:15 GMT-0400 (Eastern Daylight Time): 1 action succeeded
```

Tutorial: Specify assets in Splunk SOAR (On-premises)

This tutorial demonstrates how to run more complex actions within a playbook. In this example, you want to run actions on a specific asset. You can either specify the asset by its ID, or specify a tag to include all assets associated with that tag.

Specify assets by ID

To execute actions on specific assets, pass a list of asset IDs to the `act()` call.

```
import phantom.rules as phantom
import json

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

    if not success:
        return

    return

def on_start(incident):

    phantom.act('list vms', assets=["vmwarevsphere"], callback=list_vms_cb)
    return
```

The function generates the following result when run in the playbook debugger:

```
2015-03-14T21:12:41.365000:  Processing incident: '4' [2a76c74c-5713-11e4-8a26-9b99986c1e2a]
2015-03-14T21:12:41.369000:  act(): Action 'list vms' shall be executed on assets: vmwarevsphere
2015-03-14T21:12:41.370000:  act(): action details: [list vms] parameters: [[]] assets: [vmwarevsphere]
callback function: [list_vms_cb] and NO user specified for reviewing params
2015-03-14T21:12:41.385000:  act(): No action parameter review or asset approval requests generated.
2015-03-14T21:12:41.387000: Starting action 'list vms' on asset '28f81303-5982-451b-a833-1acdd191a763'
2015-03-14T21:12:41.410000: running: The connector 'vSphere App' started successfully. Execution parameters
sent.
2015-03-14T21:12:42.130000: running: Loaded action execution configuration
2015-03-14T21:12:42.135000: running: Connecting to 10.10.0.40...
2015-03-14T21:13:08.769000: success: 1 of 1 action succeeded
2015-03-14T21:13:08.879000: Command 'list vms' success. 1 of 1 action succeeded
2015-03-14T21:13:08.882000:  calling action callback function: list_vms_cb
 *** The Rule has completed. Result: success ***
```

Specify assets by tag

You can also pass a tag to the `act()` function. The action runs on all assets with that tag.

```
import phantom.rules as phantom
import json

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

    if not success:
        return

    return

def on_start(incident):

    phantom.act('list vms', tags=["virtual"], callback=list_vms_cb)
    return
```

By using a tag, the `list vms` action runs on all assets tagged as `virtual`.

```
2015-03-14T21:21:52.723000:  Processing incident: '4' [2a76c74c-5713-11e4-8a26-9b99986c1e2a]
```

```
2015-03-14T21:21:52.737000:  act(): Warning: For action 'list vms' no assets were specified. The action
shall execute on all matching assets
2015-03-14T21:21:52.760000:  act(): Action 'list vms' shall be executed on assets: vmwarevsphere,
vmwarevsphere2
2015-03-14T21:21:52.760000:  act(): action details: [list vms] parameters: [[]] assets: [vmwarevsphere,
vmwarevsphere2] callback function: [list_vms_cb] and NO user specified for reviewing params
2015-03-14T21:21:52.780000:  act(): No action parameter review or asset approval requests generated.
2015-03-14T21:21:52.794000: Starting action 'list vms' on asset '28f81303-5982-451b-a833-1acdd191a763'
2015-03-14T21:21:52.828000: running: The connector 'vSphere App' started successfully. Execution parameters
sent.
2015-03-14T21:21:52.833000: Starting action 'list vms' on asset '5a776fff-37d7-4a34-a299-21354dff8c45'
2015-03-14T21:21:52.863000: running: The connector 'vSphere App' started successfully. Execution parameters
sent.
2015-03-14T21:21:54.883000: running: Loaded action execution configuration
2015-03-14T21:21:54.890000: running: Connecting to 10.10.0.40...
2015-03-14T21:21:54.906000: running: Loaded action execution configuration
2015-03-14T21:21:54.912000: running: Connecting to 10.10.0.70...
2015-03-14T21:22:04.967000: success: 1 of 1 action succeeded
2015-03-14T21:22:05.097000: Command 'list vms' success. 1 of 1 action succeeded
2015-03-14T21:22:20.325000: success: 1 of 1 action succeeded
2015-03-14T21:22:20.446000: Command 'list vms' success. 1 of 1 action succeeded
2015-03-14T21:22:20.451000:  calling action callback function: list_vms_cb
 *** The Rule has completed. Result: success ***
```

Tutorial: Specify parameters in Splunk SOAR (On-premises)

Most actions require at least one parameter to function. Parameters are lists of dictionaries that are passed to the action. The specific action dictates the format and set of required parameters. Refer to the API documentation for the app you're leveraging to get the required parameters.

This example uses the WHOIS app to execute a simple WHOIS query. The WHOIS domain action requires one parameter: a domain name.

```
import phantom.rules as phantom
import json

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

    if not success:
        return

    return

def on_start(incident):

    params = [ { "domain": "phantom.us" },
               { "domain": "splunk.com" } ]

    phantom.act('whois domain', parameters=params, callback=whois_domain_cb)
    return
```

The playbook runs and produces results that you can view on the container detail screen or in Investigation in the WHOIS app.

```
2015-03-14T23:21:02.688000:  Processing incident: '4' [2a76c74c-5713-11e4-8a26-9b99986c1e2a]
2015-03-14T23:21:02.690000:  act(): Warning: For action 'whois domain' no assets were specified. The action
shall execute on all matching assets
2015-03-14T23:21:02.704000:  act(): No assets found for action 'whois domain'.
2015-03-14T23:21:02.705000:  act(): action details: [whois domain] parameters: [[{"domain": "phantom.us"},
```

```
{"domain": "splunk.com"}]] assets: [] callback function: [whois_domain_cb] and NO user specified for
reviewing params
2015-03-14T23:21:02.711000:  act(): No action parameter review or asset approval requests generated.
2015-03-14T23:21:02.712000: Starting action 'whois domain' on asset ''
2015-03-14T23:21:02.717000: running: The connector 'WHOIS App' started successfully. Execution parameters
sent.
2015-03-14T23:21:02.970000: running: Loaded action execution configuration
2015-03-14T23:21:04.845000: success: 3 of 3 actions succeeded
2015-03-14T23:21:04.864000: Command 'whois domain' success. 3 of 3 actions succeeded
2015-03-14T23:21:04.869000:  calling action callback function: whois_domain_cb
 *** The Rule has completed. Result: success ***
```

Dynamically build parameters from container data

Hard coding parameters into you scripts doesn't allow much flexibility. The key is to use data from a container and operate on it by using it as parameters to actions. You can extract data from the container itself either by directly indexing into the JSON elements or through the `collect()` call.

`collect()` uses data paths as a method to index into the JSON elements by searching for the appropriate key and retrieving the associated values. Data paths and `collect()` help simplify this.

In this example, you have an incident with some artifacts that have domain names in them within the Common Event Format (CEF) structure. You can use the following function to extract all domain names.

```
import phantom.rules as phantom
import json

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

    if not success:
        return

    return

def on_start(incident):

    params = []

    hosts = phantom.collect(incident, 'artifact:*.cef.sourceDnsDomain', 'all', 100)
    for host in hosts:
        params.append({ 'domain': host })

    phantom.act('whois domain', parameters=params, callback=whois_domain_cb)
    return
```

Example result:

```
2015-03-14T23:51:36.309000:  Processing incident: '4' [2a76c74c-5713-11e4-8a26-9b99986c1e2a]
2015-03-14T23:51:36.336000:  act(): Warning: For action 'whois domain' no assets were specified. The action
shall execute on all matching assets
2015-03-14T23:51:36.345000:  act(): No assets found for action 'whois domain'.
2015-03-14T23:51:36.345000:  act(): action details: [whois domain] parameters: [[{"domain": "phantom.us"},
{"domain": "splunk.com"}]] assets: [] callback function: [whois_domain_cb] and NO user specified for
reviewing params
2015-03-14T23:51:36.357000:  act(): No action parameter review or asset approval requests generated.
2015-03-14T23:51:36.359000: Starting action 'whois domain' on asset ''
2015-03-14T23:51:36.394000: running: The connector 'WHOIS App' started successfully. Execution parameters
sent.
2015-03-14T23:51:36.852000: running: Loaded action execution configuration
```

```
2015-03-14T23:51:38.103000: success: 3 of 3 actions succeeded
2015-03-14T23:51:38.116000: Command 'whois domain' success. 3 of 3 actions succeeded
2015-03-14T23:51:38.121000:  calling action callback function: whois_domain_cb
 *** The Rule has completed. Result: success ***
```

Tutorial: Chain a series of actions in Splunk SOAR (On-premises)

This topic demonstrates how to execute a series of actions to bind together many security operations tasks into a single playbook in Splunk SOAR (On-premises). You execute a series of actions by incorporating new actions into the callback of another action.

You don't need to use callbacks to run multiple actions in parallel if those actions have no interdependency. Instead, invoke multiple act() commands sequentially. However, if one action depends on the results of another action, the second action must be called from within the first action's callback.

Prerequisites

Before completing this tutorial, review the following topics to learn about callbacks and extracting data from a container:

- Tutorial: Specify assets in Splunk SOAR (On-premises)
- Tutorial: Specify parameters in Splunk SOAR (On-premises)

Parse results

The results of an action are passed to the callback as the fourth parameter. The results contain an array of results that you can access as a JSON structure. You can run a single action on multiple assets, in which case a result exists for each asset.

More than one parameter might have been provided to the action, meaning there is an additional array of results with one result for each parameter. In that case, the result structure can be three levels deep. If you intend to parse results in your own code, you must iterate through all assets and all parameters to access them.

For example, a simple action that calls disable user with a single user produces the following results:

```
import phantom.rules as phantom
import json

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

    phantom.debug(results)

    if not success:
        return

    return


def on_start(incident):

    users = [{ "username" : "jason_malware" }]

    phantom.act('disable user', parameters=users, assets=["domainctrl1"], callback=disable_user_cb)

    return
```

```
[
{
    'status': 'success',
    'action_results': [
        {
            'status': 'success',
            'data': [ ],
            'message': 'Userstatechanged',
            'parameter': {
                'username': 'jason_malware'
            },
            'summary': { }
        }
    ],
    'asset': 'domainctrl1',
    'summary': {
        'total_objects': 1,
        'total_objects_successful': 1
    }
}
]
```

Calling this same Action on two users, with one invalid user name, produces an array of results:

```
import phantom.rules as phantom
import json

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

    phantom.debug(results)

    if not success:
        return

    return


def on_start(incident):

    users = [{ "username" : "jason_malware" },
             { "username" : "fred_malware" }]

    phantom.act('disable user', parameters=users, assets=["domainctrl1"], callback=disable_user_cb)

    return
```

```
{
    'status': 'success',
    'action_results': [
        {
            'status': 'success',
            'data': [ ],
            'message': 'Userstatesameasrequired',
            'parameter': {
                'username': 'jason_malware'
            },
            'summary': { }
        },
        {
```

```
            'status': 'failed',
            'data': [ ],
            'message': 'SearchonADforUserDNfailed',
            'parameter': {
                'username': 'fred_malware'
            },
            'summary': { }
        }
    ],
    'asset': 'domainctrl1',
    'summary': {
        'total_objects': 2,
        'total_objects_successful': 1
    }
}
```

Chain actions

If any an Action on any one Asset with any single parameter succeeds, then the overall Action is considered to have succeeded. Individual portions of that Action may have, however, failed. If we want to now operate on the results of the disable user call, we can iterate through the result structure and look for successful results, calling enable user on each disabled user:

```
import phantom.rules as phantom
import json

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

    if not success:
        return

    for result in results:
        for user in result['action_results']:
            if 'status' in user and user['status'] == 'success':
                phantom.debug(user['parameter']['username'] + ' re-enabled')

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

    if not success:
        return

    for result in results:
        for user in result['action_results']:
            if 'status' in user and user['status'] == 'success':
                phantom.debug(user['parameter']['username'] + ' disabled')
                phantom.act('enable user', parameters=[{ "username" : user['parameter']['username'] }],
assets=["domainctrl1"], callback=enable_user_cb)

    return


def on_start(incident):

    users = [{ "username" : "jason_malware" },
             { "username" : "fred_malware" }]

    phantom.act('disable user', parameters=users, assets=["domainctrl1"], callback=disable_user_cb)

    return
```

```
2015-03-21T00:29:59.131000:  Processing incident: '4' [2a76c74c-5713-11e4-8a26-9b99986c1e2a]
2015-03-21T00:29:59.137000:  act(): Action 'disable user' shall be executed on assets: domainctrl1
2015-03-21T00:29:59.137000:  act(): action details: [disable user] parameters: [[{"username":
"jason_malware"}, {"username": "fred_malware"}]] assets: [domainctrl1] callback function: [disable_user_cb]
and NO user specified for reviewing params
2015-03-21T00:29:59.167000:  act(): No action parameter review or asset approval requests generated.
2015-03-21T00:29:59.168000: Starting action 'disable user' on asset '6c9c1b21-c259-4382-88ce-957cf6bb7a8e'
2015-03-21T00:29:59.189000: running: The connector 'LDAP App' started successfully. Execution parameters
sent.
2015-03-21T00:29:59.340000: running: Loaded action execution configuration
2015-03-21T00:29:59.367000: running: Ldap module initialized
2015-03-21T00:29:59.541000: running: Got Base DN =  'DC=corp,DC=contoso,DC=com'
2015-03-21T00:29:59.542000: running: Connecting to 10.10.0.42...
2015-03-21T00:29:59.628000: running: Got User Base DN CN=Jason Malware,CN=Users,DC=corp,DC=contoso,DC=com
2015-03-21T00:29:59.674000: running: Disabling user
2015-03-21T00:29:59.729000: running: Ldap module initialized
2015-03-21T00:29:59.883000: running: Got Base DN =  'DC=corp,DC=contoso,DC=com'
2015-03-21T00:29:59.884000: running: Connecting to 10.10.0.42...
2015-03-21T00:29:59.973000: success: 1 of 2 actions succeeded
2015-03-21T00:29:59.991000: Command 'disable user' success. 1 of 2 actions succeeded
2015-03-21T00:29:59.996000:  calling action callback function: disable_user_cb
2015-03-21T00:30:00.814000: jason_malware disabled
2015-03-21T00:30:00.823000:  act(): Action 'enable user' shall be executed on assets: domainctrl1
2015-03-21T00:30:00.824000:  act(): action details: [enable user] parameters: [[{"username":
"jason_malware"}]] assets: [domainctrl1] callback function: [enable_user_cb] and NO user specified for
reviewing params
2015-03-21T00:30:00.854000:  act(): No action parameter review or asset approval requests generated.
2015-03-21T00:30:00.855000: Starting action 'enable user' on asset '6c9c1b21-c259-4382-88ce-957cf6bb7a8e'
2015-03-21T00:30:00.871000: running: The connector 'LDAP App' started successfully. Execution parameters
sent.
2015-03-21T00:30:01.014000: running: Loaded action execution configuration
2015-03-21T00:30:01.037000: running: Ldap module initialized
2015-03-21T00:30:01.222000: running: Got Base DN =  'DC=corp,DC=contoso,DC=com'
2015-03-21T00:30:01.224000: running: Connecting to 10.10.0.42...
2015-03-21T00:30:01.328000: running: Got User Base DN CN=Jason Malware,CN=Users,DC=corp,DC=contoso,DC=com
2015-03-21T00:30:01.396000: running: Enabling user
2015-03-21T00:30:01.441000: success: 1 of 1 action succeeded
2015-03-21T00:30:01.468000: Command 'enable user' success. 1 of 1 action succeeded
2015-03-21T00:30:01.473000:  calling action callback function: enable_user_cb
2015-03-21T00:30:02.383000: jason_malware re-enabled
 *** The Rule has completed. Result: success ***
```

Develop, test, and deploy playbooks in Splunk SOAR (On-premises)

Playbooks can encode a very simple and repetitive set of simple actions OR can encode a very complex strategy to actively deal with a security breach or an incident. These strategies may be comprised of many actions combined to be executed either serially or in parallel.

Actions can be executed independent of each other (and hence in parallel) if they are called one after the other in a Playbook. However in order to execute them in sequence, either because there is a genuine dependency between two actions (parameters to action #2 are the output of action #1), action #1 has to specify a callback and in the callback of action #1, action #2 can be called.

In order to build these Playbooks and confidently deploy them, the platform supports the ability to debug them so that the author can see what the playbook is doing. Once the author is confident of the results and the Playbook is executing actions as expected, the Playbook can be saved. If the intention is to let the Playbook be executed in real time as new containers or artifacts are coming in, the Playbook has to be enabled.