IBM **Security**

IBM Resilient SOAR Platform
Function Developer’s Guide
V36

Date: February 2020

Resilient SOAR Platform
Function Developer's Guide

Platform Version	Publication	Notes
36.0	February 2020	Initial publication.

Contents

Chapter 1. Objective

This guide provides the information to integrate the Resilient Security Orchestration, Automation and Response (SOAR) Platform with your organization’s existing security and IT investments using the functions feature. Integrations makes security alerts instantly actionable, provides valuable intelligence and incident context, and enables adaptive response to complex cyber threats.

This guide is intended for programmers, testers, architects and technical managers interested in developing and testing functions with the Resilient platform. It assumes a general understanding of the Resilient platform, message-oriented middleware (MOM) systems, and a knowledge of writing scripts in Python.

Chapter 2. Overview

You should familiarize yourself with the Resilient architecture and the relevant Resilient features, as described in the following sections, before designing and writing functions.

Resilient SOAR Platform

The Resilient platform is a central hub for incident responses. It is customizable so that it can be tailored to meet the needs of your company or organization. The focus of these customizations is the dynamic playbook, which is the set of rules, conditions, business logic, workflows and tasks used to respond to an incident. The Resilient platform updates the response automatically as the incident progresses and is modified.

You should be familiar with your organization’s customized Resilient playbook when designing an integration. In particular, you should be familiar with the following playbook components:

- Rule. A set of conditional statements that identify relationships and run responses accordingly. Rules define a set of activities that are triggered when conditions are met. Activities include setting incident field values, inserting tasks into the task list, launching workflows, running internal scripts to implement business logic and placing items on message destinations to be acted upon by remote programs.
- Workflow. A graphically designed set of activities that allows you to create a complex set of operations. You can use workflows to implement sophisticated business processes that can be invoked by rules. Workflows can contain various components, such as scripts, functions, and message destinations.
- Script. For users familiar with writing Python scripts, you can write scripts to access incident data (same data as accessed by rules) then perform activities more complex than can be handled by rules. Scripts can be triggered by rules or workflows.
- Message destination. The location where data is posted and made accessible to remote programs. The message includes details about an object and the activity taken. You can configure rules, workflows and functions to send messages to one or more message destinations.
- Custom field. Design element used in incident layouts to capture specific data. An integrated system can populate a custom field.
- Data table. Design element that organizes data in a tabular format. An integrated system can populate the table.
- Function. A Resilient object that sends data to a remote function processor through a message destination. The function processor is the remote code component that performs an activity and returns the results. The results are acted upon by scripts, rules, and workflow decision points to dynamically orchestrate the security incident response activities. Functions simplify development of integrations by wrapping each activity into an individual workflow component. A function consists of the following components:
  - Inputs. Data that is acted on by the function processor. The inputs can be provided by a Resilient user or by a pre-process script.
  - Pre-process script. A script that is used to dynamically set the value of one or more of the function’s input fields. You can use the script to retrieve the current value of a property then provide that value to the function as an input. A pre-process script cannot perform write activities on objects, such as changing incident values or adding artifacts.
  - Output. Result of the function processor. A post-process script can act upon this result. If saved, objects within the same workflow instance and executed after the function can also access the data.
  - Post-process script. A script that performs an activity in response to the result provided by the function. The script can change incident values, add artifacts, add data table rows, and more.

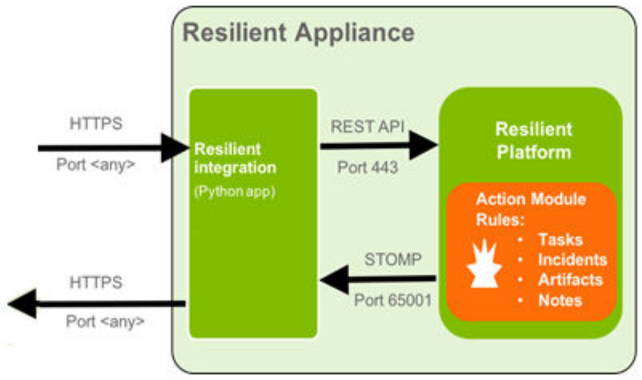
For more information about the Resilient platform and dynamic playbooks, refer to the *Playbook Designer Guide*.

Integration architecture

The Resilient platform has a full-featured REST API that sends and receives JSON formatted data. It has complete access to almost all Resilient features, including but not limited to: creating and updating incidents and tasks, managing users and groups, and creating artifacts and attachments.

To perform an integration with functions, your Resilient platform must subscribe to the Action Module. This is an extension to the Resilient platform that allows implementation of custom behaviors beyond what is possible in the Resilient internal scripting feature. It is built on Apache ActiveMQ. The STOMP message protocol is used for Python based integrations. Custom behaviors are triggered by adding a message destination to a rule defined in the Resilient platform and subscribing your integration code to that message destination.

The following diagram shows the relationship between the integration component, REST API, Action Module and Resilient platform.



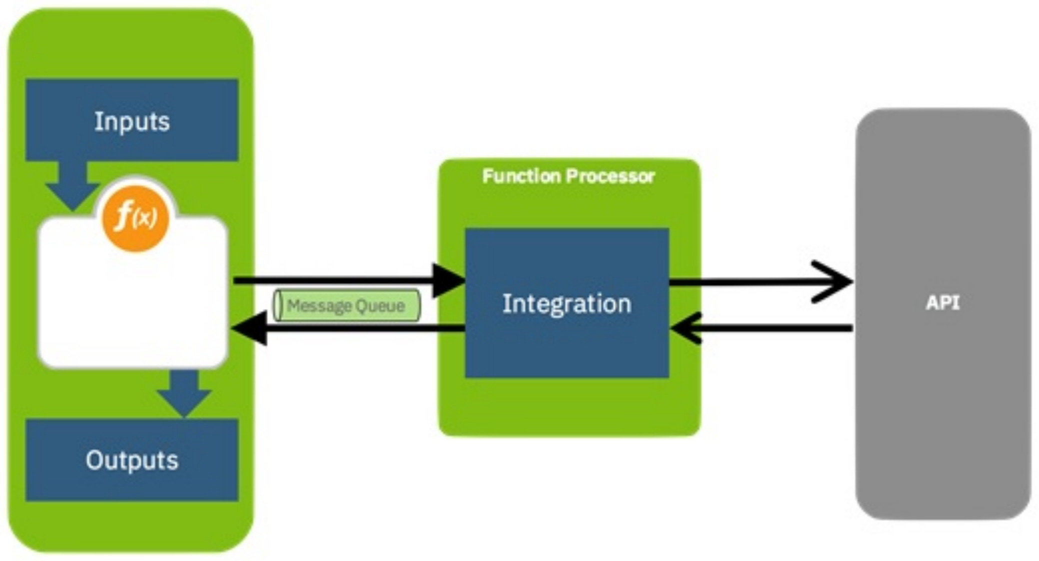
The Resilient platform contains an interactive Rest API browser that allows you to access the Resilient REST API and try out any endpoint on the system. When logged into the Resilient platform, click on your account name at the top right and select **Help/Contact**. Here you can access the complete API Reference guide, including schemas for all of the JSON sent and received by the API, and the interactive Rest API.

Function processor

Resilient functions send data to external programs called function processors. The function processor can then perform integration work, for example:

- Performing a lookup, such as for information about a user or machine in an asset database, and returning the data values found.
- Searching SIEM logs, such as for an IP address, a URL or a server name, and returning a list of events.
- Sending a file attachment to a sandbox for analysis, producing a report and a collection of observables.
- Opening a ticket in an ITSM system with a type, name and description, and returning the ticket-ID.
- Triggering an external action and then returning results for use in workflows, tasks and other decisions.

In the Resilient platform, a rule is configured on an incident, an artifact or other object. When the rule fires, it runs a workflow that could have multiple steps. Those steps can include functions that send input parameters to the function processor using a message destination, receive the results, and use the results to update the Resilient incident, to decide the direction of subsequent workflow steps or in a variety of other ways.



The Resilient Circuits framework is required and makes it extremely simple to develop and deploy functions using Python. The function processor component is a Python class that implements *function methods*. These functions are called by the framework when the Resilient platform invokes the function from a workflow.

Resilient Circuits

Resilient Circuits is a Python circuits framework that automatically manages authenticating and connecting to the STOMP connection and REST API in the Resilient platform. It simplifies creating integrations by allowing you to focus on writing the behavior logic. It is the preferred method for writing integrations.

You can use Resilient Circuits to manage your functions as well as other types of integrations. Each integration has its own section in the app configuration file. This file stores information about the Resilient platform, such as user credentials, as well as variables for your functions.

Resilient Circuits use the Resilient helper module, which is a Python library to facilitate easy use of the Rest API.

Development Overview

Before you write a function, you must understand the purpose of the function, the inputs the function needs to perform its activity, the expected results, and the actions or decisions to be made based on the results.

The following procedure provides a high-level overview of the development process. The subsequent sections in this guide provide the details.

1. Make sure you have Resilient Circuits framework installed and configured on a Resilient Integration Server, as described in the *Integration Server Guide*.
2. Log into the Resilient platform and create one or more functions and associated components, such as message destination and workflow.
3. Using Resilient Circuits and codegen, write the function processor, which is the code for the integration.

   TIP: If you have access to functions that are similar to the one you wish to create, use that function as a template to save time.

4. Use the pip install command to deploy the function to the Resilient platform and test the function by triggering the workflow and checking the results. If you make any changes to any of the function's components, repackage your function processor using codegen.

   NOTE: The Resilient Circuits codegen command automatically accesses the latest changes to the Resilient functions and associated components.
5. Write a document that provides information on deploying and using the function.
6. Package your function, all its components and the document then distribute it to other Resilient administrators. They can deploy the package to any Resilient platform at the same or later version as your test platform.
7. Optionally, share your package with other developers in the IBM Resilient developer community.

Developer website

The Resilient developer web site contains the core Resilient helper module and Resilient Circuit packages, additional integration packages, documentation and examples. The links are provided below.

- IBM Resilient Developer website. Provides overview information and access to various areas of development, such as developing playbooks and publishing integrations.
- IBM Resilient Github. Provides access to library modules, community-provided extensions, example scripts, and developer documentation. It also contains the Resilient Circuits and helper module packages. This is also accessible from the developer website reference page.
- IBM X-Force App Exchange. Provides access to the Resilient community apps on IBM X-Force.
- Releases. Lists the apps by Resilient platform release. You can also download from this page.

In addition, you can view the Resilient product guides, such as the Playbook Designer Guide, and additional information in the IBM Knowledge Center. (This link takes you to a page where you can choose the version of the Resilient platform.)

Chapter 3. Resilient Circuits framework

The procedures to install and configure the Resilient Circuits framework on a Resilient integration server are included in the *Integration Server Guide*. The guide also provides the values for the Resilient platform within the configuration file, named app.config by default.

In a development environment, you may find it necessary to override one or more values from your app.config file when running Resilient Circuits. For example, you may want to temporarily run with the log level set to DEBUG. To accomplish this, run Resilient Circuits with:

```
resilient-circuits run --loglevel DEBUG
```

You can also use optional parameters to run the application when the values being overridden are required and missing from the config file.

For a complete list of optional arguments for overrides, run:

```
resilient-circuits run -- --help
```

NOTE: If on a Windows system and you edit the file with Notepad, ensure that you save it as type **All Files** to avoid a new extension being added to the filename, and use UTF-8 encoding.

Chapter 4. Functions

Before you write a function, you must understand the purpose of the function, the inputs the function needs to perform its activity, the expected results, and the actions or decisions to be made based on the results.

If you wish, you can create multiple functions related to the same integration and include all of them in one package.

TIP: If you have access to functions that are similar to the one you wish to create, use that function as a template to save time.

Creating the Resilient platform components

Before you write the function processor code, you need to create the function and associated components as follows. For detailed procedures on creating each component, see the *Playbook Designer Guide*.

1. Review the conventions in Naming Conventions before creating the function and its components.
2. Log in to the Resilient platform as a user with permission to view and modify the customization settings.
3. Go to the Message Destinations tab and create a message destination for the function. Set the Type to **Queue** and Expect Acknowledgement to **Yes**.
4. Go to the Functions tab and create the function then define its properties:
   - **Name**: Enter a unique name that describes the purpose of the function.
   - **API Name**: Generated by the system and based on the Name field. This is the name you use when you write your function processor.
   - **Message destination**: Select the message destination you created in the previous step.
   - **Description**: Enter a description of the function's purpose (what it does), overview of the inputs (and guides to logic needed in pre-process script), and overview of outputs (and guides to logic needed in post-process script).
   - **Inputs**: One or more fields whose values are inputs for the function.
5. If you require custom fields to gather or receive specific data for your integration, perform the following:
   a. Go to the **Layouts** tab.
   b. Determine where to place the fields by selecting the tab or **New Incident Wizard**.
   c. Create the fields. Take note of the API Access name of each field for use in your code.
6. If you require a data table to receive data from the function, perform the following:
   a. Go to the **Layouts** tab.
   b. Determine where to place the data table by selecting the tab.
   c. Create the data table. Take note of the API Access name of the data table for use in your code.
7. Go to the **Workflows** tab and create or modify one or more workflows to include the function. You may wish to create an example workflow to show how the function should be used, which you can later package with your function. This is handy for functions that rely on complex data handling in pre-process or post-process scripts.

TIP: Instead of creating a new workflow, you can copy a workflow that is similar to the one you want then edit it. To copy a workflow, use the Resilient Circuits clone option, as follows:

```
resilient-circuits clone --workflow <existing_workflow_api_name> <clone_workflow_api_name>
```

8. Add the function then enter the following:
   a. Determine whether to enter the input values manually in the Input tab or use a pre-process script to determine the input values.
   b. Choose whether to save the output of a function for use in the pre- or post-process scripts for functions further in the workflow.
9. Enter the input values or enter a post-process script. In either case, see Input Field Considerations for guidelines.
10. Enter a post-process script to act upon the result. See Post-Process Script Considerations for guidelines.
11. At the Rules tab, create the rule to execute the workflow you created. Make sure that the rule does not include a message destination. The function defines the message destination. You may wish to create an example rule to show how the function should be used, which you can later package with your function.

Naming conventions

The name of your integration package should reflect the type of integration as well as be unique. Here are a few examples:

- LDAP Query
- McAfee ePO Integration
- Elasticsearch Query
- Cisco Get Domains

The names of the components in your package should reflect your package name, as follows:

- Use a unique prefix for your function and components, such as the product or provider name. Using LDAP Query as example, the components would be as follows. When you enter the name of a component, the API name replaces spaces with underscores.
  - Function name: ldap_query
  - Message destination name: ldap_query
  - Input field: ldap_query_inputname
  - Data table: ldap_query_datatable
  - Custom field: ldap_query_fieldname

  You do not need to add a prefix to commonly used fields, such as incident_id, task_id, attachment_id, and artifact_id.
- If you package rules that are meant as examples, you should use the prefix, Example, such as: “Example: Query LDAP”
- If you package workflows that are meant as examples, you should also use the prefix, Example. In your function, you use the API name of the workflow, so an example of a workflow name would be: example_ldap_query_workflow_name

Input field considerations

Consider the following when defining input fields.

- Input fields are referenced via the inputs object, for example inputs.id.
- For any ID input field, you must define the corresponding input field type as Number. Frequently used ID input fields include:

  - incident_id
  - artifact_id
  - task_id
  - attachment_id
- When sending an ID field to an input field, you must define the input field as Numeric.
- Text Areas, such as incident.description, must be passed to input fields using the 'content' property. For example: inputs.fn_description = incident.description.content
- In the Resilient platform, use the Tooltip field to provide extra information about the input. Use an example if helpful.
- Use care when processing TextArea fields, which can contain rich text. Rich text is passed to your function as HTML tags. For example, a bold word is passed as '<b>word</b>'. This can be confusing to integration applications that do not anticipate this tagging. In most cases, it makes sense to strip out these tags.
- An alternative to naming individual input fields for each parameter to pass to an integration, use a single input field containing a JSON string:

```
inputs.test_details = """{{ "incidentId": "{0}",
"name": "{1}",
"description": "{2}" }}""".format(str(incident.id), incident.name,
incident.description.content)
```

- Decoding this input field may require the removal of control characters:

```
test_details = kwargs.get("test_details")  # text

mpa = {}.fromkeys(range(32))
dict = json.loads(test_details.translate(mpa))
log.info("incidentId: "+dict['incidentId'])
```

- Binary format, such as an attachment, is not supported. If a function needs the content of an attachment, do not send it through input fields. Instead, the integration needs to call the resilient_client of its super class to get the file content. For example:

```
resilientClient = self.rest_client();
"""
     Example of call:
    /incidents/2095/artifacts/13/contents
"""
api = "/incidents/{}/artifacts/{}/contents".format(incidentID, artifactID)
response = resilientClient.get_content(api)
```

- Input fields can be a composite of multiple Resilient fields, for example:

```
inputs.jira_description =
"Incident types: {}\nNIST Attack Vectors: {}\n\nAdditional Information: {}"
   .format(incident.incident_type_ids, incident.nist_attack_vectors,
           incident.description)
```

You should test all required input parameters for a valid entry. You can also enforce this when defining the input field (Requirement: Always). Pre-process scripts are needed for field assignment.

```
incident_id = kwargs.get('incident_id')
if not incident_id:
     raise FunctionError('incident_id is required')
```

Post-process script considerations

Consider the following when defining post-process scripts.

- Use the results object to access the data returned from an integration. Due to limitations in the way Python scripts can be written, the syntax to access the JSON data can vary:
  - results.matched_list should be used rather than results['matched_list']

– results.matched_list['file'] should be used for the next level item

- Failed post-process scripts may cause a workflow to remain in the running state. For a given incident, click the **Actions** button then select **Action Status** to verify the successful completion of a function. Click the **Actions** button then select **Workflow status** to terminate any workflow with failed actions.
- Use dict.get("key") when testing whether data exists for that key. The dict["key"] may cause the script to fail if "key" does not exist.

Writing the function processor

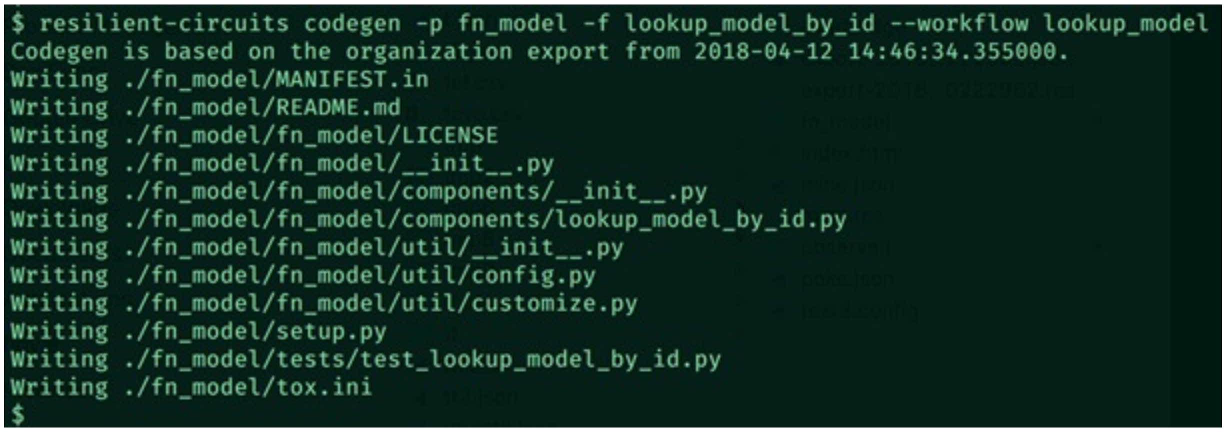
A function processor component, in this framework, is a Python class that implements *function methods*. These functions are called by the framework when the Resilient platform invokes the function from a workflow.

To write the Python code that performs the function's integration logic, start by using codegen to generate a Python package with a boilerplate implementation. This package includes everything needed to make your function installable. Besides your code, it can include the function definition, custom fields, data tables, workflows and rules, which you created on the Resilient platform.

```
resilient-circuits codegen --package pckg_name --function func_name
```

NOTE: To see additional options, such as packaging multiple functions, see Packaging your function.

The following example packages a function called lookup_mode_by_id and a workflow called lookup_model into a package called fn_model.



```
$ resilient-circuits codegen -p fn_model -f lookup_model_by_id --workflow lookup_model
Codegen is based on the organization export from 2018-04-12 14:46:34.355000.
Writing ./fn_model/MANIFEST.in
Writing ./fn_model/README.md
Writing ./fn_model/fn_model/LICENSE
Writing ./fn_model/fn_model/__init__.py
Writing ./fn_model/fn_model/components/__init__.py
Writing ./fn_model/fn_model/components/lookup_model_by_id.py
Writing ./fn_model/fn_model/util/__init__.py
Writing ./fn_model/fn_model/util/config.py
Writing ./fn_model/fn_model/util/customize.py
Writing ./fn_model/setup.py
Writing ./fn_model/tests/test_lookup_model_by_id.py
Writing ./fn_model/tox.ini
$
```

The result is a directory containing the essential files for an installable Python package that implements the function or functions specified. Within this package, the function code itself is a simple script, such as the one below, that you can edit to add your integration logic. This script is a *component* with a method, decorated with **`@function()`**that tells the Resilient Circuits framework how to call it.

```
"""Function implementation"""

import logging
from resilient_circuits import ResilientComponent, function, StatusMessage, FunctionResult,
FunctionError


class FunctionComponent(ResilientComponent):
    """Component that implements Resilient function 'lookup_model_by_id"""


    @function("lookup_model_by_id")
    def _lookup_model_by_id_function(self, event, *args, **kwargs):
        """Function: Lookup more information about the specified ID"""
        try:
            # Get the function parameters:
            model_id = kwargs.get("model_id")  # text

            log = logging.getLogger(__name__)
            log.info("model_id: %s", model_id)

            # PUT YOUR FUNCTION IMPLEMENTATION CODE HERE
            #  yield StatusMessage("starting...")
            #  yield StatusMessage("done...")

            results = {
                "value": "xyz"
            }

            # Produce a FunctionResult with the results
            yield FunctionResult(results)
        except Exception:
            yield FunctionError()
```

First, the function gets its parameters (`model_id`, in this case). The boilerplate implementation logs the values for ease of debugging, although you may want to change that if it's too noisy.

At any stage during the function's processing, you can enter `yield StatusMessage` ("..."), which provides a status message that will display to the Resilient user in the Action Status dialog. If your function might run for several seconds or minutes, this can be a useful way to show progress.

The `results` are a Python dictionary, containing named values that will be available in the workflow output and post-process script. If possible, your functions should return a small 'results' dictionary with one or a few named values. In this case, you should include enough documentation to help your users understand how to find and use these results in a custom workflow.

Modifying the selftest module

One of the boilerplate templates created when you use codegen is the selftest module. The selftest module provides the means to diagnose issues before escalating them. You can configure a run-time capability to test connectivity and configuration issues by modifying this module, which is located in a folder called selftest.py under the util directory.

One of the basic tests is a connectivity test. IBM Resilient suggests that you configure the module to perform a non-invasive test, such as a login or simple query.

You determine the results that would be useful for a self-diagnostic. The selftest result is configured to return a state of success, failure or unimplemented.

Once you complete your package, you can use the selftest command to test the function, as follows:

```
resilient-circuits selftest
```

If you have multiple packages, you can run selftest on one or more specific integrations:

```
resilient-circuits selftest -l <integration_1> <integration_n>
```

Running the function

Before running the function, you must install this package so that Resilient Circuits can load it. The most convenient way to install during development is with pip's editable flag (or `-e`). Using this, you can edit your source files directly without needing to reinstall after any changes.

```
pip install --editable ./pckg_name/
```

Run the integration code from the command-line, as follows. The framework reads your configuration file, connects to the Resilient platform, finds and loads all the installed components, then subscribes to the message destination for each function processor component. Leave it running; when a function is invoked, the code handles it.

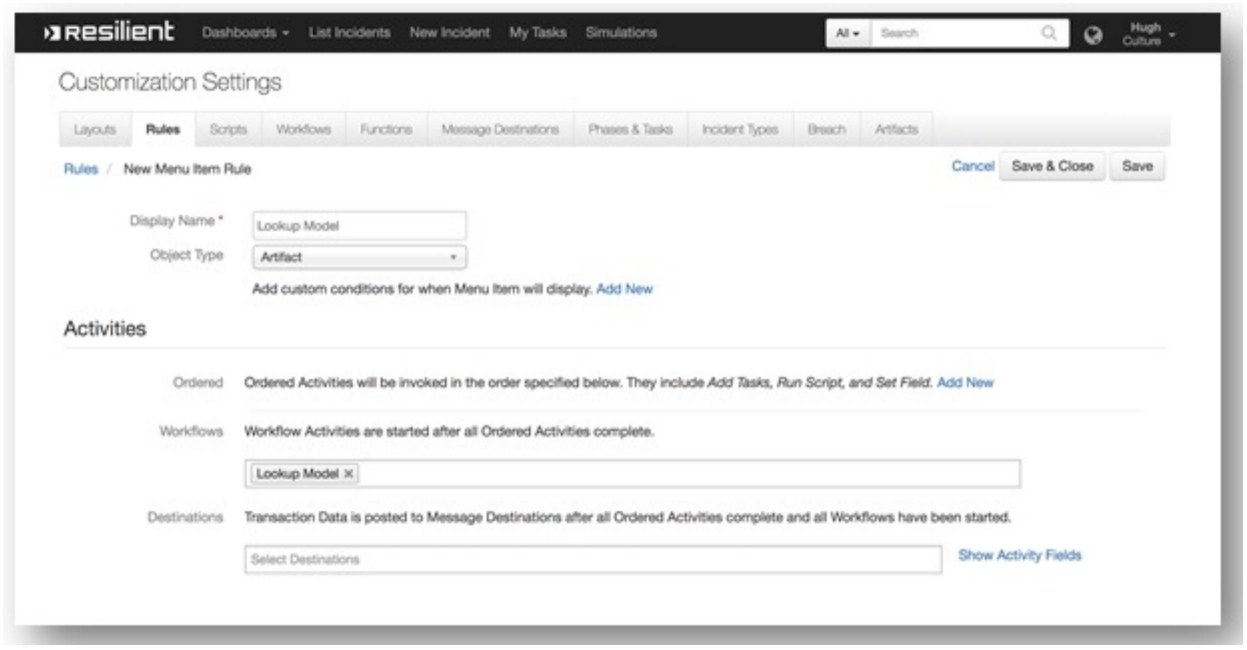
```
resilient-circuits run
```

```
2018-04-12 16:10:31,814 INFO [app] Configuration file: /home/integration/.resilient/app.config
2018-04-12 16:10:31,816 INFO [app] Resilient server: myserver.resilientsystems.com
2018-04-12 16:10:31,817 INFO [app] Resilient user: api@example.com
2018-04-12 16:10:31,818 INFO [app] Resilient org: PartnerLab
2018-04-12 16:10:31,818 INFO [app] Logging Level: INFO
2018-04-12 16:10:38,075 INFO [component_loader] Loading 1 components
2018-04-12 16:10:38,076 INFO [component_loader]
'fn_model.components.lookup_model_by_id.FunctionComponent' loading
2018-04-12 16:10:38,089 INFO [stomp_component] Connect to myserver.resilientsystems.com:65001
2018-04-12 16:10:38,090 INFO [actions_component]
'fn_model.components.lookup_model_by_id.FunctionComponent' function 'lookup_model_by_id'
registered to 'function_example'
2018-04-12 16:10:38,091 INFO [app] Components loaded
2018-04-12 16:10:38,094 INFO [app] App Started
2018-04-12 16:10:38,196 INFO [actions_component] STOMP attempting to connect
2018-04-12 16:10:38,198 INFO [stomp_component] Connect to Stomp...
2018-04-12 16:10:38,199 INFO [client] Connecting to myserver.resilientsystems.com:65001 ...
2018-04-12 16:10:38,825 INFO [client] Connection established
2018-04-12 16:10:39,090 INFO [client] Connected to stomp broker
[session=ID:ip-1-2-3-252.srv.resilientsystems.com-35733-1523282148180-5:243, version=1.2]
2018-04-12 16:10:39,092 INFO [stomp_component] Connected to failover:(ssl://
myserver.resilientsystems.com:65001)?maxReconnectAttempts=1,startupMaxReconnectAttempts=1
2018-04-12 16:10:39,093 INFO [stomp_component] Client HB: 0  Server HB: 15000
2018-04-12 16:10:39,094 INFO [stomp_component] No Client heartbeats will be sent
2018-04-12 16:10:39,095 INFO [stomp_component] Requested heartbeats from server.
2018-04-12 16:10:39,098 INFO [actions_component] STOMP connected.
2018-04-12 16:10:39,205 INFO [actions_component] Subscribe to message destination
'function_example'
2018-04-12 16:10:39,206 INFO [stomp_component] Subscribe to message destination
actions.201.function_example
```

The framework is running and waiting for the function to be called.

Testing the function

You test the function by using it from the Resilient platform. Use a rule to trigger the workflow that contains the function. The rule can be a menu-item rule, which displays an action for its object (such as incident, artifact or task) when the conditions are met, or an automatic rule that runs the workflow when an object is created or modified and meets the rule's pre-defined conditions.



Customization Settings

Rules / New Menu Item Rule

Display Name * Lookup Model

Object Type Artifact

Add custom conditions for when Menu Item will display.

Activities

Ordered: Ordered Activities will be invoked in the order specified below. They include *Add Tasks, Run Script, and Set Field.*

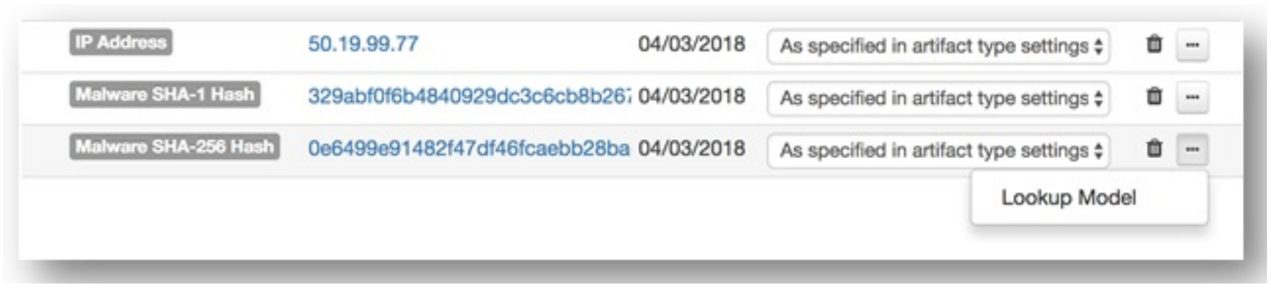
Workflows: Workflow Activities are started after all Ordered Activities complete.

Lookup Model ×

Destinations: Transaction Data is posted to Message Destinations after all Ordered Activities complete and all Workflows have been started.

Select Destinations

For this example, the workflow runs on an artifact, so the rule is a menu-item rule that appears as an action in each incident's artifact action menu, accessible from the [**...**] button.



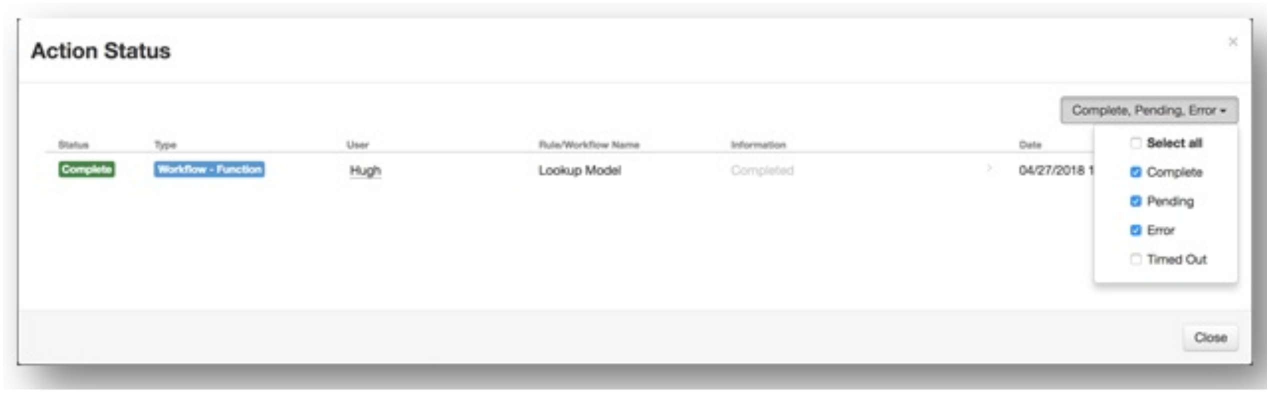
IP Address	50.19.99.77	04/03/2018	As specified in artifact type settings
Malware SHA-1 Hash	329abf0f6b4840929dc3c6cb8b26	04/03/2018	As specified in artifact type settings
Malware SHA-256 Hash	0e6499e91482f47df46fcaebb28ba	04/03/2018	As specified in artifact type settings

Select your action to start the workflow and call your function.

At the integration console, you see the function message arrive, including the logging message to print the function's parameters as part of the boilerplate code.

```
2018-04-12 16:24:23,235 INFO [actions_component] Event: <lookup_model_by_id[] (id=219,
workflow=lookup_model, user=who@example.com) 2018-04-12 16:23:20.221000> Channel:
functions.function_example
2018-04-12 16:24:23,237 INFO [lookup_model_by_id] model_id:
0e6499e91482f47df46fcaebb28bac985193aab331beeb5bc553162b422c1f21
```

The incident's Action Status menu shows the status of each rule, which can be pending (queued for delivery to the function processor), processed successfully, or with an error. In the following example, the action completed with success and included a status message.



Action Status

Status	Type	User	Rule/Workflow Name	Information	Date
Complete	Workflow - Function	Hugh	Lookup Model	Completed	04/27/2018 1

If you need to make any changes to any of the function's elements, repackage your function processor using codegen.

Consider implementing the actual integration code to the 3rd party system in a separate module from that which manages the Resilient framework. This separation of logic allows you to test the 3rd party integration separate from the code as part of Resilient Circuits.

```
components/
   <function_code>.py
   <3rd_party_integration_code>.py

tests/
   test_<function_code>.py
   test_<3rd_party_integration_code>.py
```

You can also use the Resilient Circuits sefltest command as a troubleshooting aid.

Packaging your function

When satisfied with the test results, you can package your function and components for use with other Resilient platforms. You can deploy the package to a Resilient platform in your environment different than the one you were using for testing, or make it available to the Resilient community.

Creating your package

The following is the basic command you use to create a Python package containing your function:

```
resilient-circuits codegen --package <package_name> --function <function_name>
```

When you are ready to share your package, you need to include all the related components, such as the rules, workflows, message destinations and any custom fields and data tables. To do this, use arguments such as --workflow, --rule, --m (message destination) and --datatable. Use --h to list all the arguments. The following command includes a workflow and rule, and specifies the export file:

```
resilient-circuits codegen --package <package_name> --function <name> --workflow <name> --rule
<name> --exportfile <filename>
```

NOTE: The --exportfile is needed only if you do not wish to use the most recent changes.

If using a rule or script name that contains spaces, use quotation marks around the name and be aware that the name is case sensitive. For example:

```
resilient-circuits codegen -m fn_cisco_enforcement --rule “Cisco Get Domains” --datatable
cisco_enforcement
```

Optionally, you can specify multiple functions to codegen into the package by specifying each function name with the --function parameter. To see a list of functions on your integration system, enter the following command. Codegen returns the file path to the function, which you can open using a text editor such as vi or nano.

```
resilient-circuits codegen
Available functions:
    ldap_search
    ldap_disable_user
```

NOTE: The codegen -o argument, shown as follows, should only be used if you need to create a single Python file with a specific name.

```
resilient-circuits codegen --function <function_name> -o <function_name>.py
```

If you have previously created a package and you need to recreate it, you can use the reload option. This is useful when you wish to add more components without specifying the components you added the last time you created the package. To use the reload option, you must specify the name of the package to be reloaded. The following example reloads your package with all previous components then adds a new rule.

```
resilient-circuits codegen --reload <package_name> --rule “Cisco Add Domain”
```

This command also saves the existing customize.py as customize-*yyyymmdd-hhmmss*.bak in the util directory of the package.

NOTE: You cannot reload a package generated with a version of Resilient Circuits prior to V31. If you wish to use reload, you need to rename your customize.py then generate a new package using Resilient Circuits V31 or later. You can then use reload for the new package.

Documenting your package

IBM Resilient suggests that you create a document that describes the function and all its components. It should discuss how to install it, its basic purpose, and a complete description of each component.

If your package contains custom fields and data tables, make sure to document that the Resilient playbook designer should include these in the incident layouts, since the fields and data tables cannot be automatically added to layouts. When possible, provide details on the intended incident layouts, such as a New Incident Wizard or a specific incident tab.

You should also document that re-importing a function’s custom fields and custom data tables does not restore any layout changes made. Therefore, the layout changes may need to be recreated.

Compressing and sharing your package

You can compress your integration package into a tar.gz file then distribute that file to another Resilient platform in your organization or to the IBM App Exchange. Your package should include the functions and all their components.

To create a compressed file, perform the following.

1. Locate the setup.py module. This is created as one of the boilerplate templates when you use codegen. The setup.py module contains the necessary building blocks for the python package.
2. Edit the setup.py file. The following is an example of the fields you might need to edit:

   ```
   name='fn_example',
   version='1.0.0',
   license='<MIT, APACHE, etc.>',
   author='<developer or company name>',
   author_email='<developer or company email>',
   url='<company url>',
   description="<Simple description>",
   long_description="<Longer description as necessary>",
   install_requires=[
     <list of required libraries to include when installing this package>
   ]
   ```

3. Compress the package into a tar.gz file by running:

   ```
   python setup.py sdist
   ```

   The resulting file is added to the dist/ folder using the package name and version, such as dist/fn_example-1.0.0.tar.gz.

This file can be further wrapped in a zip file for submission to the IBM App Exchange. See Publishing your function.

If you wish to distribute the package to another Resilient platform, note the following considerations:

- Copy the file to the Resilient integration server that is used with the Resilient platform.
- Make sure that the Resilient platform is at the same or greater version as the Resilient platform used to create the package.

As described in the *Integration Server Guide*, a Resilient administrator uses the Resilient Circuits customize utility to update the Resilient platform with any missing message destinations, function definitions, and other design elements. The command is:

```
resilient-circuits customize
```

Alternatively, you can install as specific package as follows:

```
resilient-circuits customize –l <package_name>
```

Each installed package that includes a “customize” entry-point is called and returns a collection of customizations described in Resilient JSON format. These are used to update the Resilient platform.

The customize command only rebuilds those components that were referenced from the codegen command line. For example, if you indicated --workflow, then workflows are rebuilt during the customize process.

Publishing your function

In addition to deploying your functions to other Resilient platforms in your environment, you can share your functions with the Resilient community.

IBM Resilient provides two locations for sharing functions, IBM Security X-Force App Exchange, and Resilient Community Apps on Github.

The IBM Security X-Force App Exchange allows you to make your function available to others in the Resilient community. You have the option to update the function as needed. For more information on submission requirements, see the Publishing Integrations page.

The Resilient Community Apps on Github also allows you to share your function source code with others. Members can copy, modify and enhance your function using the pull request mechanism. See the Resilient Community Apps page for a list of apps, with developer information at the bottom of the web page.

You can choose to submit to one or both locations.

Packaging Resilient components only

If you wish to share Resilient components, such as rules, scripts, workflows and custom fields, without deploying a function, you can package those components in a .res file using the `resilient-circuits extract` command then import the file into a Resilient platform. For example, you created a script, playbook, or workflow then determined that these components would be useful on another Resilient platform.

NOTE: The extract command does not extract functions referenced by workflows when using the `--rule` option. You need to extract the function explicitly `(-f)`.

To create a res file containing these components, perform the following:

1. Make sure that you have the Resilient export file that contains the components you wish to package.
2. Enter the following command to package the components and zip the resulting output file:

```
resilient-circuits extract --script <name> --workflow <name> --rule <name> -f <name> -o
<output_file_name>.res --zip
```

3. Copy the file to the host system of the Resilient platform.
4. Log in to the Resilient platform and use the import feature to import this file. You can find the import feature by clicking **Administrator Settings** and selecting the **Organization** tab. Refer to the *System Administrator Guide* for details.

Chapter 5. Coding considerations

The following sections provide advice and recommendations to consider when creating your function.

Data flow

Functions are blocking until results are returned. Returning `FunctionError()` aborts the result of a workflow.

All functions are stateless. No persistence of data is retained between function calls.

Error handling

The high level code should be covered with try / except / finally blocks. Any exception should describe the issue for Resilient Action Status log. The finally block should be used to perform any connection closing, temporary file cleanup, and so on.

Do not use `yield FunctionError()` within try/except. It causes the message destination message to remain stuck. Each time Resilient Circuits is restarted, the message is attempted again. The better solution is call `raise FunctionError()` and then the except block can perform the `yield`. The `yield FunctionError()` also interferes with `finally` in try/except/finally. The fix is to change `yield` to `raise`.

Logging

Log information for debugging purposes. Sensitive information should never be logged, but can be obscured. For example, user.email@example.com can become us***@example.com.

Data results

Use the function, FunctionResult(), to return the JSON for post-process script processing. A sample result should be included in the function's description field.

As much as possible, return top-level items or lists of items (or combinations):

```
{
   'item1': 'result1',
   'item2': 'result2'
 }
```

or

```

{ 'entries':
   [
      {'item1': 'result1', 'item2': 'result2'},
      {'item1': 'result3', 'item2': 'result4'}
   ]
 }
```

Links back to the integration application are supported by creating a Resilient custom field, such as a TextArea with rich text enabled. Add the custom field to an incident's Summary section for easy access. In the workflow's post-processing script, add code similar to this:

```
incident.properties.pd_incident_url = "<a href='{}' target='blank'>Link</
a>".format(results.pd['incident']['html_url'])
```

Data tables are a convenient way to return and display row results. There are a number of considerations to bear in mind with their use:

- Data tables are global to an incident (as are all custom fields). This means that the table rows are not unique to a specific object, such as an artifact.
- In order to deal with the shared use of tables, add two columns to a table to identify the search argument (such as artifact.value) and a timestamp when the results were returned.

Linking to 3rd party integration objects

Perform the following to create a live URL link back to the integrated system, such as a ticketing system.

1. Create a new incident custom field as a text area with Rich Text enabled.
2. Add it to the summary section of an incident's layout.
3. In the function's post-process script, build the URL as an HTML anchor:

   ```
   incident.properties.pd_incident_url = "<a href='{}' target='blank'>reference Link</
   a>".format(results.pd['incident']['html_url'])
   ```

Rich text

Rich text fields may contain HTML markup. This format may be undesirable for the target integration system. A method such as the one below can strip off the HTML elements, preserving some of the new line format:

```
def _cleanHtml(htmlFragment):
    '''
    Resilient textarea fields return html fragments. This routine will remove
      the html and insert any code within <div></div> with a linefeed
    :param htmlFragment:
    :return: cleaned up code
    '''

    tmp = re.sub(r'</div>', '\n', htmlFragment)
    tmp = re.sub(r'</ol>', '\n', tmp)       # numbered lists
    tmp = re.sub(r'</li>', '\n', tmp)       # unnumbered lists
    return re.sub(r'<([^>]+)>', '', tmp)       # removes all remaining html
```

Temporary files

For integrations that read from or write to files, one method to create temporary files is to use Python's tempfile:

```
import tempfile
...
with tempfile.NamedTemporaryFile() as temp_file:
```

In situations when filehandles are used, consider using StringIO instead. This avoids any OS interaction with files:

```
import StringIO
...
with StringIO(filedata) as temp_file:
```

For more information, see https://docs.python.org/2.7/library/stringio.html.

Temporary files can behave differently depending on the operating system, such as opening a file a second time while the named temporary file is still open. This works on Unix but not on Windows NT or later. To avoid this, create the temp file with the delete=False flag so the file is not deleted when closed, close the file before trying to read/write a second time and then explicitly delete the file when done rather than depending on NamedTemporaryFile to do it. The Floss function in the Resilient App Exchange contains an example of how to do this.

Avoid using self.

Using `self.` in functions is not thread-safe. This means that `self.` is shared by all functions (even though they are running in different threads). Therefore, a statement such as, `self.connection = Connection` will likely be overwritten when more than one function is invoked and running at the same time.

The better solution is to use Class level variables and only use `self.` for class function and Resilient static functions.

Chapter 6. Example: Making an API call

If using Python to access the Resilient directly, the following script shows how to authenticate using an API key account and make an API call to get a list of constants from the Resilient REST API.

```
"""
    This script uses API Key Authentication with Resilient platform to
    get list of constants from a server. Modify as needed.

    Prerequsites:
    * Python 2.7 or higher

    It is recommended that you run this in a virtualenv as follows:
    * pip install virtualenv
    * virtualenv myvirtualenvdir
    * . myvirtualenvdir/bin/activate

    Once activated, verify that the requests module is installed
    * pip install requests
"""
from __future__ import print_function
from requests.auth import HTTPBasicAuth
import requests
import json


def main():
    """
        * Resilient version must be 33.0 or higher
        * API Key ID and Secret are known. Sample values shown in script.
        * API Key has the permissions needed for the operation
    """

    key_id = "c5b0d84c-73d6-4dca-a9fb-e98870d62826"
    key_secret = "e2-5k4IT3WlVH9mfSl_g5ALBdtmVEaRsh4EEoFUs2HI"
    server = "9.70.195.31"
    resource = "rest/const"
    url = "https://{0}/{1}".format(server, resource)
    headers = {"Content-Type": "application/json; charset=UTF-8"}
    auth = HTTPBasicAuth(key_id, key_secret)

    req = requests.get(url, headers=headers, auth=auth, verify=False)
    print(json.dumps(req.json(), indent=4, sort_keys=True))


if __name__ == "__main__":
    main()
```