

IBM Resilient



Incident Response Platform Integrations

ODBC Query Function V1.0.0

Release Date: June 2018

Resilient Functions simplify development of integrations by wrapping each activity into an individual workflow component. These components can be easily installed, then used and combined in Resilient workflows. The Resilient platform sends data to the function component that performs an activity then returns the results to the workflow. The results can be acted upon by scripts, rules, and workflow decision points to dynamically orchestrate the security incident response activities.

This guide describes the ODBC Query Function.

Overview

The ODBC Query Function establishes an ODBC connection to the desired SQL database server and executes SELECT, INSERT, UPDATE or DELETE SQL statements.

This package includes a SQL script that creates a sample SQL table with dummy data, example workflows that demonstrate how to call these four SQL statements, rules that start the example workflows and a custom Resilient data table that can be updated by the SELECT statement workflow.

The remainder of this document describes the included function, how to configure it in custom workflows, and any additional customization options.

Installation

Before installing, verify that your environment meets the following prerequisites:

- Resilient platform is version 30 or later.
- You have a Resilient account to use for the integrations. This can be any account that has the permission to view and modify administrator and customization settings, and read and update incidents. You need to know the account username and password.
- You have access to the command line of the Resilient appliance, which hosts the Resilient platform; or to a separate integration server where you will deploy and run the functions code. If using a separate integration server, you must install Python version 2.7.10 or later, or version 3.6 or later, and "pip". (The Resilient appliance is preconfigured with a suitable version of Python.)

Install the Python components

The functions package contains Python components that will be called by the Resilient platform to execute the functions during your workflows. These components run in the 'resilient-circuits' integration framework.

The package also includes Resilient customizations that will be imported into the platform later.

Ensure that the environment is up to date,

```
sudo pip install --upgrade pip
sudo pip install --upgrade setuptools
sudo pip install --upgrade resilient-circuits
```

To install the package, you must first unzip it then install the package as follows:

```
sudo pip install --upgrade fn_odbc_query-<version>.tar.gz
```

Configure the Python components

The 'resilient-circuits' components run as an unprivileged user, typically named 'integration'. If you do not already have an 'integration' user configured on your appliance, create it now.

Perform the following to configure and run the integration:

1. Using sudo, become the integration user.

```
sudo su - integration
```

2. Use one of the following commands to create or update the resilient-circuits configuration file. Use `-c` for new environments or `-u` for existing environments.

```
resilient-circuits config -c
```

or

```
resilient-circuits config -u
```

3. Edit the resilient-circuits configuration file.

- a. In the [resilient] section, ensure that you provide all the information needed to connect to the Resilient platform.
- b. In the [fn_odbc_query] section, edit the settings as follows:

```
# Define your connection string
sql_connection_string=Driver={PostgreSQL};Server=IP Address;Port=5432;
Database=myDataBase;Uid=myUserName;Pwd=myPassword;

# Optional settings:

# Define restricted SQL statements as a list, separated by a comma, using
square brackets.
# Example ["delete", "update", "insert"].
# Comment this line if there are no restrictions.
sql_restricted_sql_statements=["delete", "insert", "update"]

# Define if you wish to execute commits automatically after every SQL
statement.
# Comment this line to use false - the default.
sql_autocommit=true

# Define a query timeout in seconds.
# Comment this line to use the default 0, which means "no timeout".
# Might not be supported for all database drivers.
sql_query_timeout=10
```

```
# Encoding and decoding settings needed for your SQL database.
# Define which one of supported SQL Server database settings you want to
use.
# At the moment MariaDB, PostgreSQL and MySQL are supported.
# Comment this line if you don't wish to configure decoding/encoding.
sql_database_type=MariaDB

# Define number of rows to fetch.
# Comment this line to fetch all.
sql_number_of_records_returned=10

# Some ODBC drivers might throw an error while setting
db_connection.timeout.
# Psqldb driver (PostgreSQL) throws a general error 'HY000'
# Override this SQLSTATE if your odbc driver is throwing a different
error.
sql_pyodbc_timeout_error_state=HY000
```

Connecting to a database

The ODBC function uses an open source Python module, pyodbc, and an ODBC driver to connect to a data source. ODBC drivers are database-specific and are typically written by the manufacturer of the database. More information on pyodbc and ODBC drivers is available on the [GitHub Pyodbc Wiki](#).

Connections to databases are made through the use of connection strings, which are driver-specific. General connection string information for most databases is available at <http://www.connectionstrings.com>.

Standard PostgreSQL connection string:

```
Driver={PostgreSQL};Server=IP address;Port=5432;Database=myDataBase;
Uid=myUsername;Pwd=myPassword;
```

Standard MySQL connection string:

```
Server=myServerAddress;Port=3306;Database=myDataBase;Uid=myUsername;
Pwd=myPassword;
```

Unicode configuration

The pyodbc module recommends configuring ODBC connection's Unicode encoding and decoding settings that are specific for the chosen database and the version of Python in use.

ODBC Function V1.0.0 supports Unicode settings for MariaDB, PostgreSQL and MySQL databases using Python 2.7.

Users may implement additional support by downloading this [function](#) and editing it as shown in the following figure.

```
# These databases tend to use a single encoding and do not differentiate between
# "SQL_CHAR" and "SQL_WCHAR". Therefore you must configure them to encode Unicode
# data as UTF-8 and to decode both C buffer types using UTF-8.
# https://github.com/mkleehammer/pyodbc/wiki/Unicode
if sql_database_type in SINGLE_ENCODING_DATABASES:
    db_connection.setdecoding(pyodbc.SQL_CHAR, encoding='utf-8')
    db_connection.setdecoding(pyodbc.SQL_WCHAR, encoding='utf-8')
    if sys.version_info[0] == 3: # Python 3.x
        db_connection.setencoding(encoding='utf-8')
    else:
        db_connection.setencoding(str, encoding='utf-8')
        db_connection.setencoding(unicode, encoding='utf-8')
```

More information on Unicode settings is available on [GitHub Pyodbc Wiki](#).

Deploy customizations to the Resilient platform

The package contains function definitions that you can use in workflows, and includes example workflows and rules that show how to use these functions.

Deploy these customizations to the Resilient platform with the following command:

```
resilient-circuits customize
```

Answer the prompts to deploy functions, message destinations, workflows and rules.

Run the integration framework

To test the integration package before running it in a production environment, you must run the integration manually with the following command:

```
resilient-circuits run
```

The resilient-circuits command starts, loads its components, and continues to run until interrupted. If it stops immediately with an error message, check your configuration values and retry.

Configuration of resilient-circuits for restartability

For normal operation, resilient-circuits must run continuously. The recommend way to do this is to configure it to automatically run at startup. On a Red Hat appliance, this is done using a systemd unit file such as the one below. You may need to change the paths to your working directory and app.config.

The unit file should be named 'resilient_circuits.service':

```
sudo vi /etc/systemd/system/resilient_circuits.service
```

The contents:

```
[Unit]
Description=Resilient-Circuits Service
After=resilient.service
Requires=resilient.service

[Service]
Type=simple
User=integration
WorkingDirectory=/home/integration
ExecStart=/usr/local/bin/resilient-circuits run
Restart=always
TimeoutSec=10
Environment=APP_CONFIG_FILE=/home/integration/.resilient/app.config
Environment=APP_LOCK_FILE=/home/integration/.resilient/resilient_circuits.lock

[Install]
WantedBy=multi-user.target
```

Ensure that the service unit file is correctly permissioned:

```
sudo chmod 664 /etc/systemd/system/resilient_circuits.service
```

Use the systemctl command to manually start, stop, restart and return status on the service:

```
sudo systemctl resilient_circuits [start|stop|restart|status]
```

Log files for systemd and the resilient-circuits service can be viewed through the journalctl command:

```
sudo journalctl -u resilient_circuits --since "2 hours ago"
```

Function Description

Once the function package deploys the function, you can view it in the Resilient platform Functions tab, as shown below. The package also includes example workflows and rules that show how the function can be used. You can copy and modify these workflows and rules for your own needs.

In order to try out the included example workflows, users can populate the database with provided SQL script MOCK_DATA.SQL located in /doc/data directory.

Customization Settings

Layouts	Rules	Scripts	Workflows	Functions	Message Destinations	Phases & Tasks	Incident Types
---------	-------	---------	-----------	------------------	----------------------	----------------	----------------

[Functions](#) / fn_odbc_query

Name *

API Name * ⓘ

Message Destination *

Description

fn_odbc_query

fn_odbc_query

fn_odbc_query ▼

A function that runs ODBC queries. Parameters are passed to the database separately, protecting against SQL injection attacks.

Inputs

sql_query ×

sql_condition_value1 ×

sql_condition_value2 ×

sql_condition_value3 ×

ODBC Query: fn_odbc_query

A function that runs ODBC queries. The query and the data/parameters are passed to the database separately, protecting against SQL injection attacks.

This function takes the following input fields:

- **sql_query**: Query to perform. The field type is “Text with value String” and it contains four predefined example SQL statements that you can choose and edit from within the workflow. The predefined queries, as shown below, contain parameters using a question mark as a placeholder in the SQL query. Users can set values for the question marks in input fields **sql_condition_value1**, **sql_condition_value2** and **sql_condition_value3** in the workflow.
- **sql_condition_value1**, **sql_condition_value2** and **sql_condition_value3**: Parameters used in the query.

Edit Input Field

What type of field is this? ⓘ

Text with value string

API Access Name * ⓘ

sql_query

Requirement ⓘ

Always

Tooltip ⓘ

Predefined SQL statement

Placeholder ⓘ

A placeholder value

Templates

Search...



+ Row

Display Value ⓘ	Data Value ⓘ	
INSERT PostgreSQL	INSERT into mock_data (id, first_name, last_name) values (?, ?, ?)	
DELETE PostgreSQL	DELETE from mock_data WHERE id = ?	
UPDATE PostgreSQL	UPDATE mock_data SET id = ? WHERE name = ?	
SELECT PostgreSQL	SELECT id AS sql_column_1, first_name AS sql_column_2, last_name AS sql_column_3 FROM mock_data WHERE id = ?	

Displaying 1 - 4 of 4

This function implements three 'sql_condition_value' input fields. You can implement additional input fields to set more than three values for the question marks by downloading this [function](#) and editing it as shown in the following figure.

```
# -----  
# When adding more condition input fields to the function, you need to load them here  
# and pass the new variable/s to the function_utils.prepare_sql_parameters().  
# -----  
sql_condition_value1 = kwargs.get("sql_condition_value1") # text  
sql_condition_value2 = kwargs.get("sql_condition_value2") # text  
sql_condition_value3 = kwargs.get("sql_condition_value3") # text  
  
LOG.info(u"sql_condition_value1: %s", sql_condition_value1)  
LOG.info(u"sql_condition_value2: %s", sql_condition_value2)  
LOG.info(u"sql_condition_value3: %s", sql_condition_value3)  
  
sql_params = function_utils.prepare_sql_parameters(sql_condition_value1, sql_condition_value2,  
                                                    sql_condition_value3)
```


Example ODBC SELECT PostgreSQL Workflow

The “Example ODBC SELECT PostgreSQL” workflow (Object Type = Artifact) calls the ODBC query function. The Input tab of this function is shown in the following figure.

When defining the SQL table column names that the SELECT query will return, it is important to use alias column syntax. Alias names need to match Resilient data table field names. This ensures that query results are saved in the proper Resilient data table field.

The screenshot shows the 'Input' tab of a workflow configuration window. It features a table with two columns: 'Input Parameter' and 'Value'. The first row has 'sql_query' as the parameter and a text box containing the SQL query: 'SELECT id AS sql_column_1, first_name AS sql_column_2, last_name AS sql_column_3 FROM mock_data WHERE id = ?'. Below this, there are three rows for 'sql_condition_value1', 'sql_condition_value2', and 'sql_condition_value3', each with an empty text box for input.

Input Parameter	Value
sql_query	SELECT id AS sql_column_1, first_name AS sql_column_2, last_name AS sql_column_3 FROM mock_data WHERE id = ?
sql_condition_value1	
sql_condition_value2	
sql_condition_value3	

Users may insert data using the sql_condition_value1, sql_condition_value2 and sql_condition_value3 parameters on the Input tab, or set them in the Pre-Process Script to the value of the artifact associated with this workflow as shown in the following figure.

The screenshot shows the 'Pre-Process Script' tab of the same workflow configuration window. It includes a header with 'Language: Python', 'Theme' set to 'light', 'Mode' set to 'Default', and 'Tab Size' set to '2'. Below this, a code editor shows a single line of Python code: 'inputs.sql_condition_value1 = artifact.value'.

```
1 inputs.sql_condition_value1 = artifact.value
```

A Menu Item rule called “Example ODBC SELECT PostgreSQL” is also included. This rule calls the provided workflow.

When a user selects this rule from the Actions button on an incident, the rule activates the ODBC Query function. The query results update the custom “sql_query_results_dt” Resilient data table, shown in the following figure. The definition of this data table is also included in the package.

The first two data table fields are set automatically. The ‘sql_artifact_value’ field is set to the value of the artifact associated with this workflow. The ‘sql_timestamp’ field is set to the date and time of the function call. The ‘sql_column_1’ to ‘sql_column_5’ data table fields are updated by the query results.

Artifacts

Edit

Add ArtifactTableGraph

Search...

Artifact Type: AllDate Created: All ▾Has Attachment: All

Show

25

 entries

Type	Value	Created	Relate?	Actions
File Name	8	05/25/2018	As specified in artifact type settings ▾	<div><div></div><div>...</div></div>

SQL query results

Export

sql_artifact_value	sql_timestamp	sql_column_1	sql_column_2	sql_column_3	sql_column_4	sql_column_5
6	05/23/2018 14:01:18	6	Titus	Leggon	tleggon4@illinois.edu	Male
6	05/24/2018 08:45:38	6	Titus	Leggon	tleggon4@illinois.edu	Male

Displaying 1 - 2 of 2

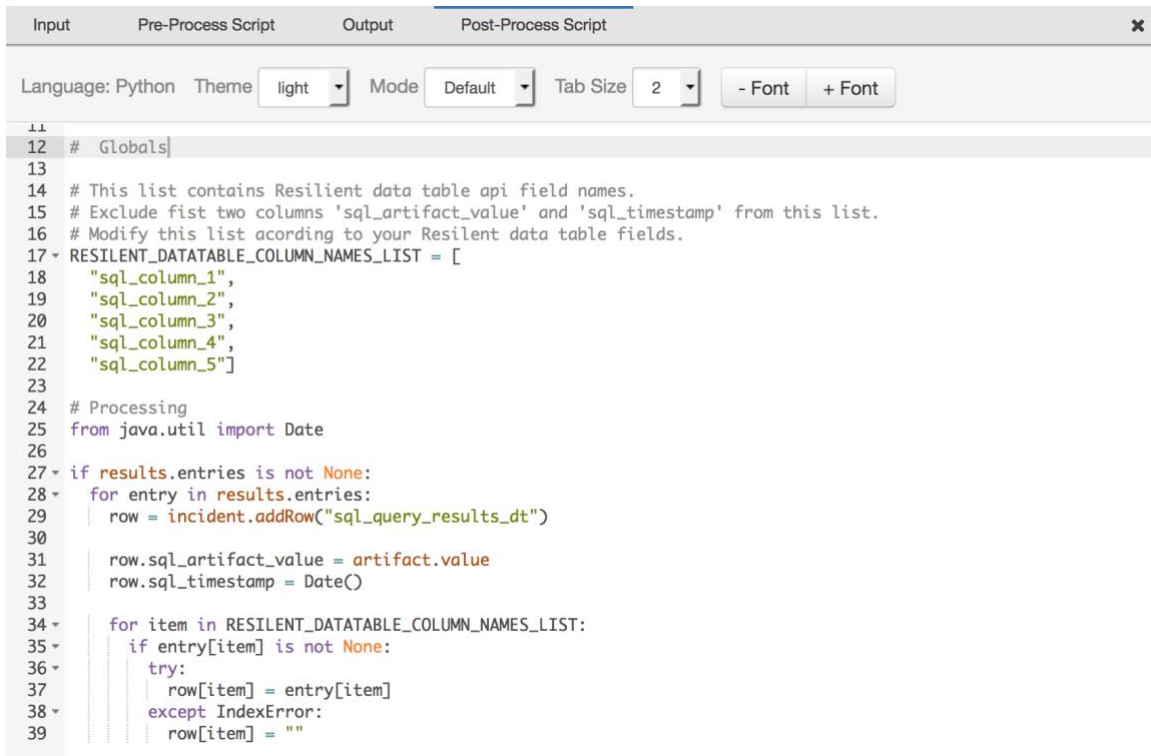
Example ODBC DELETE PostgreSQL

Example ODBC INSERT PostgreSQL

Example ODBC SELECT PostgreSQL

Example ODBC UPDATE PostgreSQL

Users may modify the “sql_query_results_dt” data table by adding or removing fields. To ensure SQL query results are saved in proper Resilient data table fields, users need to modify the RESILIENT_DATATABLE_COLUMN_NAMES_LIST in the Post-Process Script as shown in the following figure.



```
11
12 # Globals
13
14 # This list contains Resilient data table api field names.
15 # Exclude fist two columns 'sql_artifact_value' and 'sql_timestamp' from this list.
16 # Modify this list according to your Resilient data table fields.
17 RESILIENT_DATATABLE_COLUMN_NAMES_LIST = [
18     "sql_column_1",
19     "sql_column_2",
20     "sql_column_3",
21     "sql_column_4",
22     "sql_column_5"]
23
24 # Processing
25 from java.util import Date
26
27 if results.entries is not None:
28     for entry in results.entries:
29         row = incident.addRow("sql_query_results_dt")
30
31         row.sql_artifact_value = artifact.value
32         row.sql_timestamp = Date()
33
34     for item in RESILIENT_DATATABLE_COLUMN_NAMES_LIST:
35         if entry[item] is not None:
36             try:
37                 row[item] = entry[item]
38             except IndexError:
39                 row[item] = ""
```

Example ODBC UPDATE PostgreSQL Workflow

The “Example ODBC UPDATE PostgreSQL” workflow (Object Type = Artifact) calls the ODBC query function. The Input tab of this function is shown in the following figure.

Input Parameter	Value
sql_query * ⓘ	UPDATE mock_data SET id = ? WHERE name = ?
sql_condition_value1	
sql_condition_value2	
sql_condition_value3	

Users may insert data using parameters `sql_condition_value1`, `sql_condition_value2` and `sql_condition_value3` on the Input tab, or set them in the Pre-Process Script to the value and description of the artifact associated with this workflow as shown in the following figure.

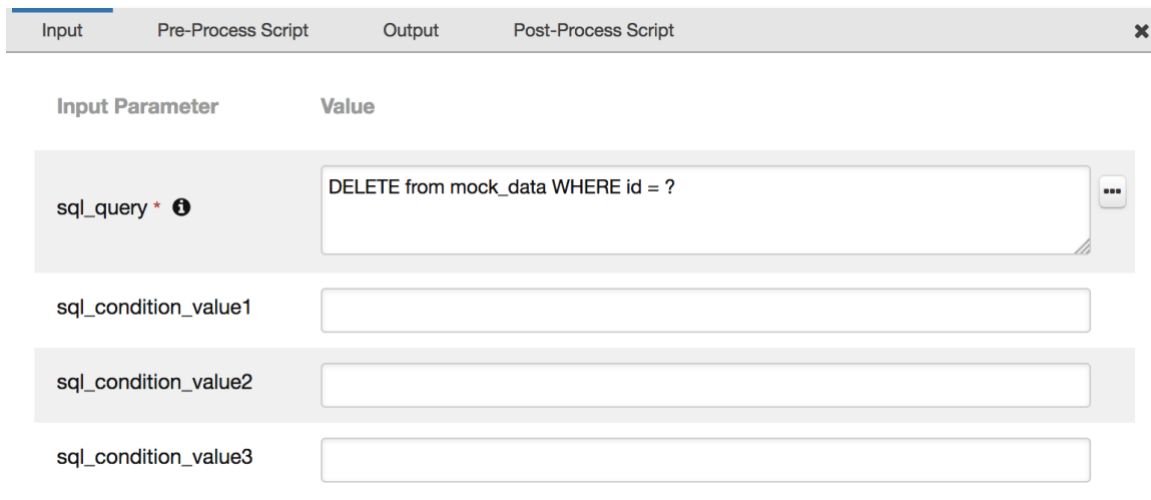
```
1 inputs.sql_condition_value1 = artifact.value
2 inputs.sql_condition_value2 = artifact.description
```

A Menu Item rule called “Example ODBC UPDATE PostgreSQL” is also included. This rule calls the provided workflow.

When a user selects this rule from the Actions button on an incident, the rule activates the ODBC Query function, which then returns the number of processed rows and displays the message in the Action Status.

Example ODBC DELETE PostgreSQL Workflow

The “Example ODBC DELETE PostgreSQL” workflow (Object Type = Artifact) calls the ODBC query function. The Input tab of this function is shown in the following figure.



Input Parameter	Value
sql_query * ⓘ	DELETE from mock_data WHERE id = ?
sql_condition_value1	
sql_condition_value2	
sql_condition_value3	

Same as the “Example ODBC SELECT PostgreSQL” workflow, users may insert data using parameter sql_condition_value1 on the Input tab, or set it in the Pre-Process Script to the value of the artifact associated with this workflow.

A Menu Item rule called “Example ODBC DELETE PostgreSQL” is also included. This rule calls the provided workflow.

When a user selects this rule from the Actions button on an incident, the rule activates the ODBC Query function, which then returns number of processed rows and displays the message in the Action Status.

Example ODBC INSERT PostgreSQL Workflow

The “Example ODBC INSERT PostgreSQL” workflow (Object Type = Artifact) calls the ODBC query function. The Input tab of this function is shown in the following figure.

Input Parameter	Value
sql_query * ⓘ	INSERT into mock_data (id, first_name, last_name) values (?, ?, ?)
sql_condition_value1	9
sql_condition_value2	Jane
sql_condition_value3	Doe

Same as the “Example ODBC SELECT PostgreSQL” workflow, users may insert data using parameters `sql_condition_value1`, `sql_condition_value2` and `sql_condition_value3` on the Input tab, or set them in the Pre-Process Script to the value of the artifact associated with this workflow.

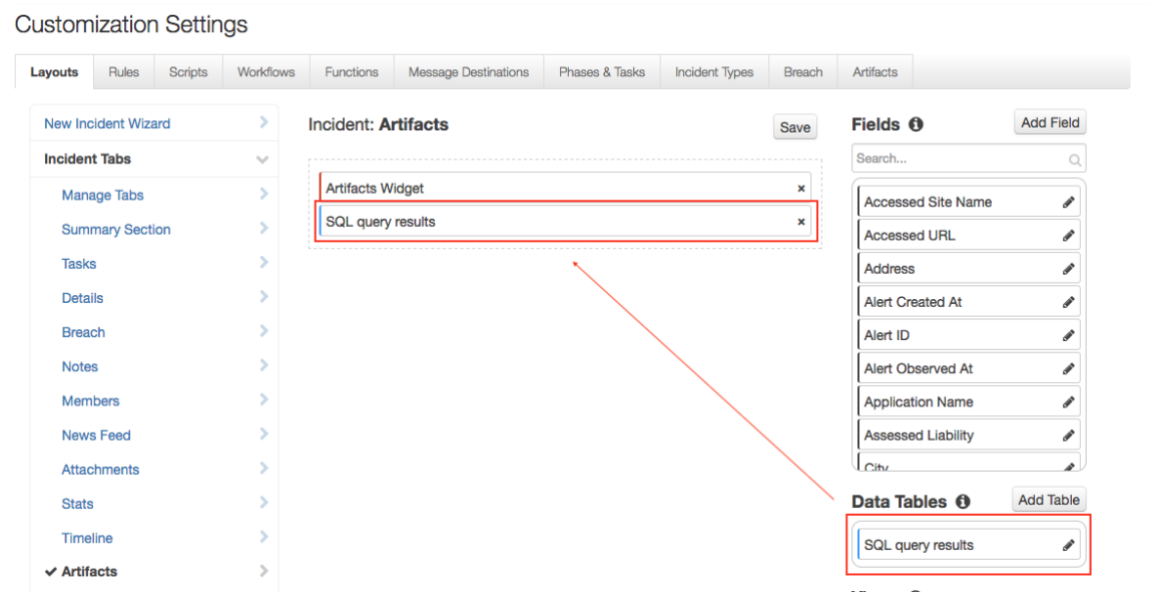
A Menu Item rule called “Example ODBC INSERT PostgreSQL” is also included. This rule calls the provided workflow.

When a user selects this rule from the Actions button on an incident, the rule activates the ODBC Query function, which then returns the number of processed rows and displays the message in the Action Status.

Resilient Platform Configuration

To display query results, users need to manually add the “SQL query results” data table to a new or existing layout.

1. Navigate to the Customization Settings and select or create a new Incident tab in the Layouts tab.
2. Drag the “SQL query results” data table to your Incident tab.
3. Click Save.



Troubleshooting

There are several ways to verify the successful operation of a function.

- Resilient Action Status

When viewing an incident, use the Actions menu to view Action Status. By default, pending and errors are displayed. Modify the filter for actions to also show Completed actions. Clicking on an action displays additional information on the progress made or what error occurred.

- Resilient Scripting Log

A separate log file is available to review scripting errors. This is useful when issues occur in the pre-processing or post-processing scripts. The default location for this log file is:

`/var/log/resilient-scripting/resilient-scripting.log`.

- Resilient Logs

By default, Resilient logs are retained at `/usr/share/co3/logs`. The `client.log` may contain additional information regarding the execution of functions.

- Resilient-Circuits

The log is controlled in the `.resilient/app.config` file under the section `[resilient]` and the property `logdir`. The default file name is `app.log`. Each function will create progress information. Failures will show up as errors and may contain python trace statements.

Support

For additional support, contact support@resilient-systems.com.

Including relevant information from the log files will help us resolve your issue.