

IBM Resilient



Incident Response Platform

QUERY RUNNER USER GUIDE v28

Licensed Materials – Property of IBM

© Copyright IBM Corp. 2010, 2017. All Rights Reserved.

US Government Users Restricted Rights: Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Resilient Incident Response Platform IBM Query Runner User Guide

Version	Publication	Notes
28.0	July 2017	Initial release.

Table of Contents

1. Overview	4
2. Check Prerequisites	5
3. Install the Integration	6
4. Create and Edit the Configuration File	7
5. Features and Behavior	8
5.1. Query Definition File Format	8
5.2. Rendering Templated Values.....	12
6. Configure the Query Integrations	13
6.1. LDAP/Active Directory	13
6.1.1. Configuration.....	13
6.2. REST API Query.....	15
6.2.1. Configuration.....	15
6.2.2. Rendering Templates.....	16
6.3. Splunk.....	17
6.3.1. Configuration.....	17
6.3.2. Rendering Templates.....	18
6.3.3. Constructing Splunk Queries.....	18
6.4. QRadar	20
6.4.1. Configuration.....	20
6.4.2. Rendering Templates.....	21
6.4.3. Constructing Ariel Queries	21

1. Overview

This document describes how to install and utilize the Resilient Query Runner integration to run queries and similar types of requests against remote systems, and update an incident with the results.

The Query Runner is a framework for triggering queries to run on remote systems triggered by Resilient rules. It installs a base set of functionality for other query integrations to use. These query integrations, installed separately, run specific types of queries. The set of query integrations currently includes:

- rc-ldap-search. Run queries against Active Directory or other LDAP services.
- rc-qradar-search. Run Ariel queries against QRadar.
- rc-query-rest. Issue simple requests against a REST API that returns JSON formatted responses.
- rc-splunk-search. Run Splunk searches.

The results of these queries are mapped into Resilient incidents. Specifically, the integrations support the following use cases:

- Use a menu item rule to manually trigger a query, and substitute values from the incident, artifact and/or activity fields into the query parameters.
- Use an automatic rule to trigger a query when conditions are met, and substitute values from the incident into the query parameters.
- Update incident fields based on query results.
- Add or update data table rows based on query results.
- Add query results to an incident as file attachments, artifacts, tasks, or notes.

2. Check Prerequisites

Verify that your environment meets the following requirements:

- Resilient platform version is 27 or later.
- Action Module is enabled on the Resilient platform.

Perform the following steps before starting the installation:

- Create or designate a Resilient user account with permission to create incidents for use by the integration.
- Choose where to install Query Runner. The integration is typically installed on the same system as the Resilient appliance; however, you can install the integration on a Unix-based or Windows operating system. If you choose a system other than the Resilient appliance, check the following:
 - System has Python version 2.7 (2.7.9 or later is recommended), or 3.4 or later. If using the Splunk search integration, you must use Python V2.7.
 - System has network access to the Resilient appliance.
- Download the latest release of the following packages.
 - co3 helper module: <https://github.com/Co3Systems/co3-api/releases/latest>
 - resilient_circuits: <https://github.com/Co3Systems/co3-api/releases/latest>
 - rc-query-runner: <https://github.com/Co3Systems/resilient-api-examples/releases/latest>
- Click [here](#) to download the desired query integration packages (one or more) to install with Query Runner. The packages include:
 - rc-ldap-search
 - rc-qradar-search
 - rc-query-rest
 - rc-splunk-search

3. Install the Integration

Perform the following steps to install the integrations:

1. Use ssh to connect to your system or virtual appliance where the installers are located.
2. Go to the folder where the installers are located.
3. Update your pip version using this command. If this updates the version of pip, you may need to log out of the shell and log back in, to ensure that the new version of pip is on the PATH.

```
sudo pip install --upgrade pip
```

4. Update or install the setuptools package using this command:

```
sudo pip install --upgrade setuptools
```

5. Install the co3 and resilient_circuits packages following the instructions in the Resilient Python integrations [README](#).
6. Install the rc-query-runner package:

```
sudo pip install -U rc-query-runner-<version number>.tar.gz
```

7. Install the query integration packages that you previously downloaded.

```
sudo pip install <filename>tar.gz
```

4. Create and Edit the Configuration File

The configuration file defines essential configuration settings for all resilient-circuits components running on the system, including all query-runner based components.

1. Use one of the following commands to create or update the configuration file.

- If you already have an app.config file being used by other integrations, you can update it to include the sections for your new query-runner based packages with:

```
resilient-circuits config -u <optional file path>
```

- If this is a new resilient-circuits installation, generate a config file with the following command. It creates the config file in your user's home directory at ~/.resilient/app.config.

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

- If you need to generate the config file to an alternate location or with a non-standard filename, you can specify the full path of the config file. Afterwards, you need to store this path to an environment variable, APP_CONFIG_FILE.

```
resilient-circuits config -c <path/filename>
```

2. Create a directory where the resilient-circuits logfiles are to be stored. This path needs to be set in the app.config file.
3. Manually edit the config file to configure the credentials to your Resilient platform organization and any remote systems. You should also configure various other parameters relating to SSL certification verification, log file path, etc.

For a complete description of all available parameters in the [resilient] section of the app.config file, please refer to the *Resilient Python Integration Development Guide*.

Query Runner itself has no associated parameters in the config file, but the various query integrations that utilize it do have parameters. These are described in [Configure the Query Integrations](#).

5. Features and Behavior

Each query requires the following:

- A rule on the Resilient platform.
- A query definition file, whose name must be the same as the rule's api-name.

In the Resilient platform, a rule can send a message to a *message destination*, which is the location where the message is stored and made accessible to external scripts, such as these integrations. Within the message, there is an action name, which is the api-name of the rule that sent the message. A rule's api-name is the same as the rule's name shown in the Resilient platform, except spaces are replaced by underscores and all characters are converted to lowercase.

For example, if you have a rule called "Splunk User Query", the following is true:

- Rule api-name is: splunk_user_query
- Action name in the message is: splunk_user_query
- Query definition file name must be: splunk_user_query

When the integration starts, it connects to the Resilient platform, listens to a specific message destination, and looks for a query definition file with the same name as rule api-name.

Each of the query definition integrations listen to different message destinations, and all rules for that integration should be associated to that single destination. The app.config section for each integration specifies the message destination, as well as the path to its query definitions files.

Once a query execution completes, the integration updates the incident with the results, as specified in the query definition's mapping rules.

5.1. Query Definition File Format

A query definition file is in JSON format. It should be stored in the query_definitions directory specified in the app.config file and named to match the associated rule in the Resilient platform.

The general format of this file, shown below, is the same across the various integrations. It consists of a "query" section that has a templated query or request to run and then one or more result behavior sections that specifies how to store the results to the incident. The structure of most sections is the same for all query-runner integrations, but additional parameters to the "query" section vary by integration. Only the "query" section is required in a definition file, all other sections are optional.

```
{
  "comments": "<description or purpose of this query>",
  "vars": {
    "<var_1>": "<string to be rendered for use in query or mapping>",
    "<var_2>": "<string to be rendered for use in query or mapping>"
  },
  "query": {
    "expression": "<query to run>",
    "extract_results_from": "results",
    "additional_queries": [
      "<additional query file to run for this action>",
      "<additional query file to run for this action>"
    ],
    "default": {
      "Result": "No data was returned from this query"
    }
  },
}
```



```

    "onerror": {
      "Result": "Query errored. No results."
    },
    "incident_fields": {
      "<field1>": "{{ row.<column 1>|js }}",
      "<field2>": "{{ row.<column 2>|js }}"
    },
    "artifacts": [
      {
        "value": "{{ row.<column 3>|js }}",
        "type": "<artifact type>",
        "description": "<description>"
      }
    ],
    "datatables": [
      {
        "name": "<data_table_name>",
        "row_id": "{{row.id}}",
        "keys": [
          "<col1>",
          "<col2>"
        ],
        "cells": {
          "<col 1>": {
            "value": "{{result.column1|js}}"
          },
          "<col 2>": {
            "value": "{{artifact.value|js}}"
          },
          "<col 3>": {
            "value": "{{result.column2|js}} {{result.column4|js}}"
          }
        }
      }
    ],
    "attachment": {
      "name": "Result_File",
      "keys": [
        "<col 1>",
        "<col 2>",
        "<col 4>"
      ],
      "ext": "<file extension to use>"
    },
    "tasks": [
      {
        "name": "<task name template>",
        "description": "<task description template>",
        "instr_text": "<task instructions template>"
      }
    ],
    "notes": [
      {
        Text"<note text template>"
      }
    ]
  }
}

```

The “**comments**” value is optional. It is a place to record a description for the purpose or intended use of the query.

The **“expression”** is the query to run. You can use Jinja2 syntax to substitute in values from incident fields; for example, `{{ incident.discovered_date }}`. Custom incident fields are accessed like `{{ incident.properties.my_custom_field|js }}`. Activity fields are accessed like `{{ properties|value('my_action_field')|js }}`. All incident values need to be valid Json in order to be submitted successfully to the Resilient REST API. Using the “js” filter, as shown, ensures that any reserved characters are escaped appropriately.

The **“vars”** section is optional. It contains a mapping of variable names to string values. These variables are rendered before the query is run, and so can be used when constructing the query. Include a variable in a query expression like `“search {{ var_name }}”`.

The **“additional_queries”** list is optional. If you want a single Resilient rule to trigger multiple queries, you can list those additional query definition files here. The additional queries are executed after results are processed from the current query.

The **“limit”** value is optional. It specifies the maximum number of search results to retrieve.

The **“extract_results_from”** should be set to the outer-most parameters in the query result JSON. Typically, this is the string “results”. If “extract_results_from” is included, then the data is accessed from a member called “result” when you do the incident or artifact mapping. If extract_results_from is not included, then the entire search result list is available all at once for mapping.

The **“default”** value is optional. If included, that object is returned as the query result if no actual results were found.

The **“onerror”** value is optional. If included, that object is returned as the query result if the query failed with an error.

The **“incident_fields”** section is a mapping specifying the incident fields that should be updated from the search result. The value for the field uses Jinja2 syntax to specify data from the query result. If a key was specified for “extract_results_from”, then the value is rendered with a single object called “result” that contains all columns from the first row of data returned in the list. Any other returned rows are ignored. If it was not included, then all of the objects from the result structure are available when the value is rendered.

The **“artifacts”** section specifies how to create an artifact from the search results. Assuming an “extract_results_from” value was specified in the query definition, then the application iterates over the list of rows and one artifact is created for each row item. The columns accessed from an object are called “result”.

The **“datatables”** section specifies a list of Resilient data tables for which you want to add or update rows from the search results.

- The **“name”** is the name of the data table.
- The **“row_id”** should be included if this action should update the same data table row that the custom action was triggered from.
- The **“keys”** list is optional. It indicates which columns uniquely distinguish a row in a data table. If **keys** is omitted and a data table row is found that matches on the key fields, then that row is updated. If **keys** is omitted or no matching row is found, then a new row is added to the table.
- The **“cells”** section is a mapping of data table column names to the value that should be used for creating or updating a row.

The “**attachment**” section specifies how to create a CSV file from the search results to upload to the incident as an attachment.

- The “**name**” value is combined with a timestamp to create the filename.
- The “**keys**” list is optional and lists which columns in the search results to include in the CSV file. If excluded, all columns are included. Including a keys list is more efficient even if you want all the columns, as it eliminates the need to walk the result list twice to determine the column names.
- The “**ext**” value is optional and lets you specify a file extension to use. If excluded, “.csv” is used for the filename.

The “**tasks**” section specifies how to create one or more tasks from the search results. Assuming an “extract_results_from” value was specified in the query definition, then the application iterates over the list of rows and one task is created for each row item. If “extract_results_from” is excluded, then one task is mapped from the entire collection of result rows. The “tasks” value should map to a list of TaskDTO objects suitable for passing to the Resilient REST API.

The “**notes**” section includes a list of one or more templated notes to create from the search results. Assuming an “extract_results_from” value was specified in the query definition, then the application iterates over the list of rows and one note is created for each row item. If “extract_results_from” is excluded, then one note is mapped from the entire collection of result rows. The “notes” value should map to a list of templated strings, each of which should render to a note text suitable for passing to the Resilient REST API.

5.2. Rendering Templated Values

The queries and query actions inside a query definition are all treated as JINJA templates. JINJA is a simple, but powerful templating language. You can read more about the syntax here <http://jinja.pocoo.org/docs/2.9/templates/>. Some of these filters require admin or master admin privileges on the Resilient integrations account you are using.

The JINJA library uses filters to format or modify values before rendering them. It includes a set of built-in filters that are always available for use, as described in <http://jinja.pocoo.org/docs/2.9/templates/#builtin-filters>.

The rc-query-runner package makes additional JINJA filters available for you to use with the templates in your query definition files, listed below. The resilient-circuits package adds a list of filters, which are documented separately.

Filter	Description	Sample Usage
user	Get a FullUserDTO object from a Resilient account email address.	{{ incident.owner user }}
value	Get the text value for an incident Select or MultiSelect field.	{{ incident.properties.my_select value }}
properties_value	Get the value for an activity field from a menu item rule.	{{ properties value("some_activity_field") }}
artifact_type	Get the Artifact Type from the type_info parameter.	"ip": "{% if type_info artifact_type == 'IP Address' %}{{ artifact.value }}{% endif %}"
prefix	Get the substring of a value from the start to the first occurrence of a substring.	"scheme": {{ incident.properties.url prefix('.') }}
suffix	Same as prefix, but gets the substring from after the first occurrence of a substring to the end of the value.	"mail_service": {{ incident.properties.email_address suffix('@') }}
fmt	For tuple/list/dict type values. Same as the built-in JINJA 'format' filter, but with the format string passed in as an argument and the substitution values coming from your value.	<pre>{{ "%s - %s" format("Hello?", "Foo!") }}</pre> <p>-> Hello? - Foo!</p> <p>This one works the other way around, so</p> <pre>{{ artifact.value fmt("Hello %s") }}</pre> <p>-> Hello Foo</p>
split	Convert a string to a list of strings using a delimiter.	{{ incident.properties.some_csv_field split(',') }}
zip	Filter applied to a tuple or list parameter. Creates a dictionary with a key from each item in the list paired with a value from the corresponding index in the passed value list.	{{ incident.members zip(("first", "second", "third")) }}
dict	Same as zip filter, but with the dict keys coming from the passed in list and the values coming from the parameter	{{ incident.members dict(("first", "second", "third")) }}

Some query-runner packages add additional JINJA filters specific to the data formats they are dealing with.

6. Configure the Query Integrations

As described in [Overview](#), the Query Runner is a platform for other query integrations. The following sections describe how to configure each of these integrations.

6.1. LDAP/Active Directory

This integration installs a Resilient circuits component called LDAPIncidentUpdate. It enables incident enrichment from user account information retrieved from an LDAP search.

6.1.1. Configuration

The LDAPIncidentUpdate component requires a section in the app.config file called [AD]. After installing the rc-ldap-search package, you can use the “resilient-circuits config –u” command to automatically add this section to an existing config file. Once added, you need to update the [AD] section to match your system configuration and desired options.

Parameter	Required?	Description and Example
queue	Y	Name of the Resilient message destination to monitor. queue = ldap
query_definitions_dir	Y	Absolute path to directory containing query definitions files. The initial value is set to an installed directory containing one or more sample query definitions. This should be changed to a directory you create containing your actual query files. query_definitions_dir = /path/to/querydefs
server	Y	LDAP host. server = ldap.mycompany.com
port	N	Port to connect to LDAP host. Default is 389. port = 389
user	Y	User account to authenticate to LDAP. user = uid=admin,cn=users,cn=accounts,dc=demo1,dc=freeipa,dc=org
password	Y	Password for the LDAP account. password = MyPassword123
auth	N	Type of authentication to use. One of: ANONYMOUS, SIMPLE, SASL, NTLM. Default is ANONYMOUS. auth = SIMPLE
ssl	N	Determines whether to use SSL when connecting to LDAP. Default is false. ssl = True
search_base	N	Location in the directory information tree where the search starts. This value can be referenced in the query definition files or they can have the value embedded directly in them. search_base = dc=demo1,dc=freeipa,dc=org

Parameter	Required?	Description and Example
log_level	N	Determines the level of log messages for this query integration. Set this to DEBUG for verbose logging. Default is INFO. log_level=DEBUG

The query definition files for use with the LDAP integration use the same general format as the other query runner based integrations. However, an additional parameter is required in the query section, called “parameters”.

```
"query": {
  "expression": "<query to run>",
  "extract_results_from": "entries",
  "parameters": {
    "search_base": "{{ options.search_base }}"
  },
  "additional_queries": [
    "<additional query file to run for this action>",
    "<additional query file to run for this action>"
  ],
  "default": {
    "Result": "No data was returned from this query"
  }
},
```

The “**parameters**” value is required for the LDAP integration. It should contain a single value called “search_base”, which is the location in the directory information tree where the search starts. In the previous example, it is pulling the value to use from the app.config file.

The **extract_results_from** parameter should always be set to “entries” for LDAP.

6.2. REST API Query

This integration installs a Resilient circuits component called QueryREST. It enables you to update incidents from the results of querying a simple HTTP/REST endpoint. Compatible endpoints expect to send and receive JSON data. While Basic Auth is supported, authentication requiring multiple requests (i.e., sessions) is not supported.

6.2.1. Configuration

The QueryREST component requires a section in the app.config file called [rest]. After installing the rc-query-rest package, you can use the “resilient-circuits config -u” command to automatically add this section to an existing config file.

Parameter	Required?	Description and Example
queue	Y	Name of the Resilient message destination to monitor. queue = rest
query_definitions_dir	Y	Absolute path to directory containing query definitions files. The initial value is set to an installed directory containing one or more sample query definitions. This should be changed to a directory you create containing your actual query files. query_definitions_dir = /path/to/querydefs
verify	N	Indicates whether to verify the certificates when request is against HTTPS endpoint. Defaults to false. verify = true
log_level	N	Determines the level of log messages for this query integration. Set this to DEBUG for verbose logging. Default is INFO. log_level=DEBUG

The query definition files for use with the REST integration use the same general format as the other query runner based integrations. However, the “vars” and “query.expression” sections contain slightly different data than in the other query-runner based integrations.

The **“extract_results_from”** parameter should not be used in REST query definition files. Instead, the entire response is accessed from a **“result”** value when mapping.

```
"vars": {
  "http-headers": {
    "X-Foo": "Bar"
  },
  "http-method": "<GET, POST, etc>",
  "http-body": {
    "artifact": "{{ artifact.value|js }}"
  }
},
"query": {
  "expression": "<URI>",
  "additional_queries": [
    "<additional query file to run for this action>"
  ],
  "default": {
    "Result": "No data was returned from this query"
  }
},
"onerror": {
  "Result": "Query errored. No results."
}
},
```

The **“vars”** section can be used in the traditional manner, for substitution in the **“query.expression”** value. There are several additional values that can be defined in the **“vars”** section that have special meaning to the QueryREST component.

The **“http-headers”** object is optional. It contains a collection of header values to set when creating the request. Among other uses, this is where authentication parameters are set. The content-type may need to be set here as well.

The **“http-method”** parameter is optional, and defaults to GET if not specified. This should be one of the valid HTTP request methods: GET, PUT, POST, PATCH, or DELETE

The **“http-body”** parameter is optional. It should render to the message body to include in the request, usually a POST or PUT call.

The **“expression”** should render to the endpoint to which you are making the request; for example, **“https://example.com/test”**.

6.2.2. Rendering Templates

All of the standard JINJA filters described here are available for use. The rc-query-rest integration adds an additional filter, **b64encode**, to enable the base 64 encoding of username/password pairs required when setting the Authorization header for Basic Auth.

Example:

```
"http-headers": {
  "Authorization": "Basic {{ (options.user, options.pass)|join(':')|b64encode }}"
}
```


6.3. Splunk

This integration installs a Resilient circuits component called SplunkQueryRunner. It enables you to trigger Splunk searches from Resilient rules and update incidents with the results. This can be used in conjunction with incidents escalated via the Resilient Splunk App (available on Splunkbase) or independently.

The account specified in your app.config to use to authenticate to Splunk must have the proper capabilities enabled to allow access to the Splunk REST API. You can read about required roles/permissions in the [Splunk REST guide](#) in the “Authentication and authorization” section.

6.3.1. Configuration

The SplunkQueryRunner component requires a section in the app.config file called [splunk]. After installing the rc-splunk-search package, you can use the “resilient-circuits config –u” command to automatically add this section to an existing config file. This adds default content to the config file, which you need to update to include your Splunk account credentials.

Parameter	Required?	Description and Example
queue	Y	Name of the Resilient message destination to monitor. queue = splunk
query_definitions_dir	Y	Absolute path to directory containing query definitions files. The initial value is set to an installed directory containing one or more sample query definitions. This should be changed to a directory you create containing your actual query files. query_definitions_dir = /path/to/querydefs
host	Y	Splunk host to connect to and run searches. Do not include the scheme (http, https). verify = example.com/splunk
port	Y	Splunk API port. This is not the port you use to connect to the Splunk UI. port = 8089
user	Y	Splunk account name for authentication. user = exampleuser
password	Y	Password for the Splunk account. password = MyPassword
query_timeout	N	Time in seconds to wait for Splunk queries to complete before giving up. Defaults to 600. query_timeout = 600
log_level	N	Determines the level of log messages for this query integration. Set this to DEBUG for verbose logging. Default is INFO. log_level=DEBUG

The query definition files for use with the Splunk integration use the same general format as the other query runner based integrations. In addition, the “query” section can include a “limit” parameter that limits the number of results the query returns. It is advised that this value is always set to a relatively low number to avoid adding hundreds of artifacts or rows to a data table.

```
"query": {
  "expression": "<query to run>",
  "extract_results_from": "results",
  "limit": 10,
  "additional_queries": [
    "<additional query file to run for this action>"
  ]
},
```

The “**limit**” value specifies the max number of query results to return. While optional, it is recommended to include it always. If mapping results to incident fields, you can set the limit to 1 since only the first result returned is used. When adding artifacts, data table rows, or any other update type that iterates over the returned results, the limit should be set to a reasonably low number so as not to flood an incident with too many artifacts, data table rows, etc.

The **extract_results_from** parameter should always be set to “results” for Splunk.

6.3.2. Rendering Templates

All of the standard JINJA filters described here are available for use. The rc-splunk-search integration adds several additional filters useful for constructing Splunk queries or converting the results.

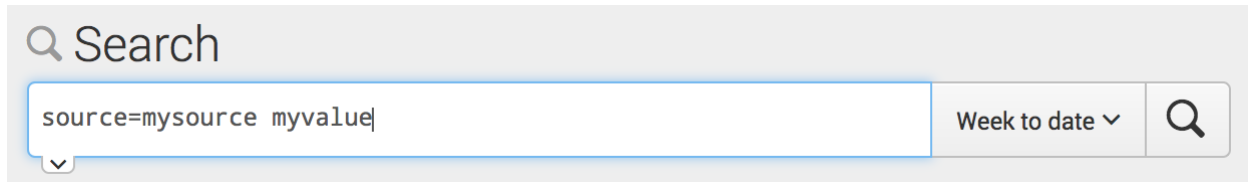
Filter	Description	Sample Usage
datetime	Converts epoch time in ms to datetime stamp in format mm/dd/YYYY:H:M:S. This is useful for using Resilient datetime fields in Splunk search time modifiers.	# Search logs for 24 hours preceding incident "expression": "search * latest={{ incident.discovered_date datetime }} earliest={{ (incident.discovered_date - (24 * 60 * 60 * 1000)) datetime }} fields *",
ms	Get epoch time in ms from a timestamp in the format YYYY-MM-DD HH:MM:SS.mmm. This is useful for creating Resilient datetime fields from timestamps in Splunk results, notably the _time field.	"incident_fields": { "my_timefield": "{{ result_time ms }}" },
mstz	Get epoch time in ms from a timestamp in the format YYYY-MM-DDTHH:MM:SS.mmm+00:00. This is useful for creating Resilient datetime fields from timestamps in Splunk results.	"incident_fields": { "my_time": "{{ result.timestamp mstz }}" },

6.3.3. Constructing Splunk Queries

The best practice for creating Splunk queries to use in your query definition files is to first proof them out in the Splunk UI and ensure that they are returning the desired results. To reduce the risk of permission issues appearing later, it is important to do this query testing using a Splunk account with the same capabilities and access as the account that your integration will use.

Your query can (and should) specify a time limiter so that it only searches data from the appropriate time period. The syntax for doing this can be found in the [Splunk documentation](#).

Once you have a query, there are some slight changes that need to be made for it to run successfully when run via the REST API. For example, you have a search that you want to run from the Resilient platform, and you want “myvalue” to come from an artifact.

A screenshot of a search interface. At the top left is a magnifying glass icon followed by the word 'Search'. Below this is a search bar containing the text 'source=mysource myvalue'. To the left of the search bar is a small dropdown arrow. To the right of the search bar is a button labeled 'Week to date' with a downward arrow, and further right is a search button with a magnifying glass icon.

This query, inside a query definition file, would look like this:

```
"expression": "search source=mysource {{ artifact.value }} earliest=@w0  
latest=now | fields *"
```

The following were done:

- Since this is a splunk search, add “search” to the start of the query. This is the most common type of query. Other types of queries might start differently, like “| rest”.
- Move your time modifiers to the equivalent query parameters.
- Use JINJA syntax to substitute values as needed.
- Specify which fields from the query results you want returned. Use an asterisk (*) to return all of them.

6.4. QRadar

This integration installs a Resilient circuits component called AQLIncidentUpdate. It enables you to trigger Ariel Queries from Resilient rules and update incidents with the results. This can be used in conjunction with incidents escalated via the Resilient QRadar App (available on XForce App Exchange) or independently.

You need to create an authorized service token in QRadar for the integration to authenticate with the QRadar REST API. This can be done in QRadar under Admin>User Management>Authorized Services. If you are using the Resilient App for QRadar, you can use the same token if you desire. This token is stored in your app.config file.

6.4.1. Configuration

The AQLIncidentUpdate component requires a section in the app.config file called [ariel]. After installing the rc-qradar-search package, you can use the “resilient-circuits config -u” command to automatically add this section to an existing config file. This adds default content to the config file, which you need to update to include your service token and preferences.

Parameter	Required?	Description and Example
queue	Y	Name of the Resilient message destination to monitor. queue = splunk
query_definitions_dir	Y	Absolute path to directory containing query definitions files. The initial value is set to an installed directory containing one or more sample query definitions. This should be changed to a directory you create containing your actual query files. query_definitions_dir = /path/to/querydefs
qradar_url	Y	QRadar host. qradar_url = https://qradar.example.com
qradar_service_token	Y	QRadar Authorized service token for authentication. qradar_service_token = XXXX-XXXX-XXXX-XXXX-XXXX
polling_interval	N	Check query status at this interval, in seconds. Defaults is 5. polling_interval = 10
query_timeout	N	Time, in seconds, to wait for ariel queries to complete before giving up. Defaults to 600. query_timeout = 600
qradar_verify	N	Should SSL certificates of the QRadar appliance be verified? Defaults to true. qradar_verify = true
log_level	N	Determines the level of log messages for this query integration. Set this to DEBUG for verbose logging. Default is INFO. log_level=DEBUG

The query definition files for use with the QRadar Ariel integration use the same general format as the other query runner based integrations. There are a few modifications to the “query” section specific to QRadar.

```
"query": {
  "expression": "<query to run>",
  "extract_results_from": "events",
  "range": "0-0",
  "additional_queries": [
    "<additional query file to run for this action>"
  ],
  "default": {
    "Result": "No data was returned from this query"
  }
},
"onerror": {
  "Result": "Query errored. No results."
}
},
```

The “**range**” value specifies the range of the results to return, with 0 based indexing. For example, if you always want only the 2nd row returned from a query, set range to “1-1”. Range is an optional parameter and for most use cases is not necessary. It is not a substitute for specifying a LIMIT value in your query.

The **extract_results_from** parameter should be set to the ariel database table you are querying. Typically, “events” or “flows”.

6.4.2. Rendering Templates

All of the standard JINJA filters described here are available for use. The rc-qradar-search integration adds an additional filter useful for constructing Ariel queries.

Filter	Description	Sample Usage
datetime	<p>Converts epoch time in ms to datetime stamp in format YYYY-MM-DD HH:mm:ss.</p> <p>This is useful for using Resilient datetime fields as values for the START and STOP parameters in queries.</p>	<p># Get time range from activity fields</p> <pre>"expression": "select * from events where INOFFENSE({{incident.properties.qradar_id}}) LIMIT 1 START '{{ properties.start_time datetime }}' STOP '{{ properties.end_time datetime }}'" </pre>

6.4.3. Constructing Ariel Queries

The best practice for creating Ariel queries to use in your query definition files is to first proof them out in the QRadar UI and ensure that they are returning the desired results, with acceptable query performance. To reduce the risk of permission issues appearing later, it is important to do this query testing using an account with the same capabilities and access as the token that your integration will use.

Guidance for the Ariel query language can be found in the [IBM Knowledge Center](#).

It is important to always include a relatively low LIMIT in your queries to prevent excessive data table rows or artifacts being created in your incident. Once you have a query, swap out any necessary values to substitute with JINJA syntax and copy it into your query definition file.