Shell Actions

This project lets you run shell-scripts (PowerShell, bash, etc.) from custom actions in Resilient.

Project Overview

The project runs as a standalone Python application. When this application starts, it connects to the Resilient platform and starts listening for actions. A manual or automatic custom action sends a message to the application. The application then locates a shell script, based on the name of the action, and runs it. Parameters to the script are taken from the properties of the incident, artifact, or other object where the action was taken. The results of the script are then attached to the incident, or used to enrich the incident in other ways.

Installation

Install the project onto a Windows or Unix machine. This machine requires

- Python version 2.7.6 or later, and 'pip'
- Network access to the Resilient appliance via ports 443 and 65001
- Resilient Systems application version 24 or later, with Action Module.

Python Libraries

You must install several Python libraries that are required by the application.

First, install the "co3" module, which is the Resilient Systems REST API client library for Python; and the "resilient_circuits" module, which is the Resilient action module application framework for Python.
Follow the installation instructions here:

`https://github.com/Co3Systems/co3-api/tree/master/python`

Basic Configuration

First, edit the 'app.config' file and supply values needed to connect to your Resilient server. You will need the hostname, port, user credentials that the application will use to authenticate, and the name of the user's Organization.

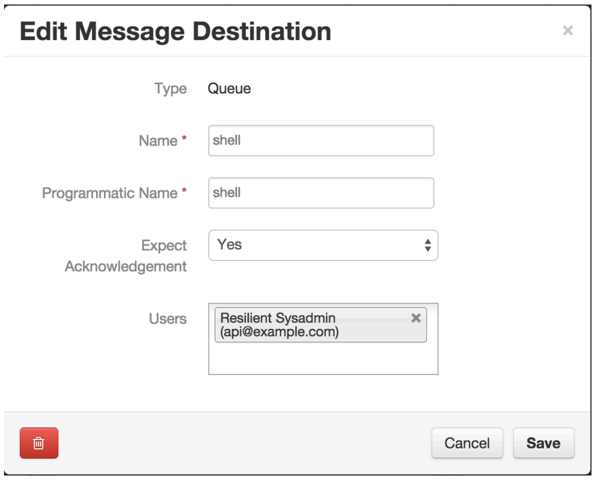
For the user credentials, we recommend creating an "API user account" specifically for integration purposes, distinct from normal users. Note that any edits that this integration makes to incidents in Resilient (adding attachments, etc.) will be attributed to this user.

In the "[shell]" section, edit the paths for the default PowerShell or Bash script handling, following the instructions.

The project includes a pair of trivial example scripts (‘shell_example.ps1’ for PowerShell, and ‘shell_example.sh’ for Bash). To run these from Resilient, you must configure the custom actions as follows:

Message Destination

In Resilient, open Administrator Settings → Actions → Message Destinations.
Add a message destination, type “Queue”, with programmatic name “shell”.
Add your API user account to the Users list for this message destination.

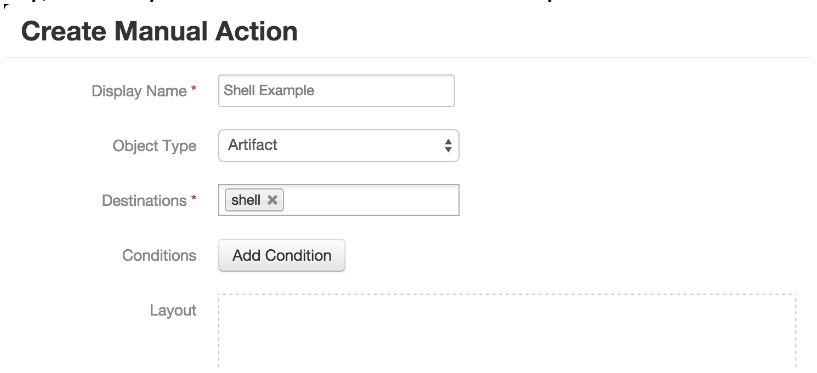


Edit Message Destination

Type	Queue
Name *	shell
Programmatic Name *	shell
Expect Acknowledgement	Yes
Users	Resilient Sysadmin (api@example.com)

Custom Action

In Resilient, open Administrator Settings → Actions → Manual Actions.
Add a manual action, name “Shell Example”.
Select object type “Artifact”.
Select your “shell” message destination.
Optionally, add any custom action fields to the layout.



Create Manual Action

Display Name *	Shell Example
Object Type	Artifact
Destinations *	shell
Conditions	Add Condition
Layout	

Note that the name of the action (“Shell Example”) is used to find a script (“shell_example”) in the ‘scripts’ directory of the application.

Running the Application

Follow the configuration instructions above to set appropriate values into the ‘app.config’ file, to create your custom actions, and to configure your shell scripts in the ‘scripts’ directory.

Then, start the application, using:

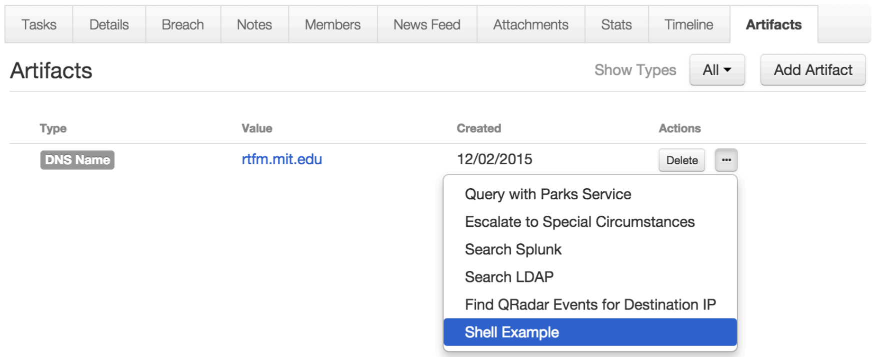
```
python run.py
```

The application will start, log some messages, and wait for action messages.

```
$ python run.py
INFO app.py Configuration file is app.config
INFO connectionpool.py Starting new HTTPS connection (1): resilient
INFO app.py App Started
INFO actions_component.py STOMP connected
INFO actions_component.py Component <Shell/actions.shell 51251:MainThread
(queued=0) [S]> registered to actions.shell
INFO actions_component.py Subscribe to 'shell'
INFO component_loader.py Loaded component 'shell_runner'
INFO app.py Components loaded
```

On Windows, the application can be configured to run as a service; instructions are not included here, but are available on request.

In the Resilient application, open an incident, and choose your custom action “Shell Example” from the “Actions” menu of an artifact:



When the action is executed, the application log will show the shell script being executed,

```
INFO shell_runner.py shell: shell_example
INFO shell_runner.py Run: bash
/home/resilient/actions/scripts/shell_example.sh "rtfm.mit.edu"
(/tmp/tmp2MV7hZ)
```

The output of the shell script is added to the incident, as a new attachment.

Snort Examples

The package includes three example scripts to manage Snort IPS rules. These show slightly more complex script behavior:

- Appending an artifact value to the end of a rule file on a remote host, directly (for blacklist/whitelist reputation rules) or with some formatting (for a general Snort rule),
- Running snort in 'test mode' (on the remote host) to produce a log showing that the rule is active,
- Sending Snort the SIGHUP signal to tell it to reload its rules.

To run 'add_snort_rule.sh', create a manual Artifact action named "Add Snort Rule".

To run 'add_to_snort_blacklist.sh', a manual Artifact action named "Add to Snort Blacklist". This should be conditional on artifact type "IP Address".

To run 'add_to_snort_whitelist.sh', a manual Artifact action named "Add to Snort Whitelist". This should be conditional on artifact type "IP Address".

These scripts must be edited with your own settings to specify the snort server, rule file and other parameters. For configuration details, refer to the 'app.config' and to the contents of each shell script.

The "disposition" for these rules is to write the output of the shell scripts into a new Note in the incident.

Framework Details

The description below assumes a Manual Action. The same features can also be used from an Automatic Action in an identical manner, but without access to Action Fields.

When a custom action is defined (Administrator Settings → Actions → Manual Action) with a message destination set to “shell”, this action will be executed by the shell action processor.

Specific actions can be configured to specific commands. See later Configuration section for details. The default is as follows.

The action name is lowercased, and non-alphanumeric characters are converted to underscores. If a file with this name and .ps1 extension exists in the “scripts” folder, it is executed. The script is given access to all the context of the action (the incident; the task or artifact or other object that the action was launched from; the user who executed the action). The output from the script is attached to the incident as a new file attachment.

Input Parameters

By default, if this action was initiated from an artifact, the script receives a single command-line parameter: the value of the artifact. So a script can declare that parameter and use it. For example:

```
Param(
  [string]$ value
)
Write-Output $value
```

Environment Context

The script is run with an environment variable EVENTDATA. This contains the file path of a temporary file that contains all the event context, in JSON format. This context file contains a top-level dictionary (hash) with values that include

- “incident”: fields of the incident, including custom fields, which are nested below “properties”
- “artifact”: fields of the artifact, if this action was triggered from an artifact,
- “task”: fields of the artifact, if this action was triggered from a task,
- “milestone”: fields of the artifact, if this action was triggered from a milestone,
- “note”: fields of the artifact, if this action was triggered from a note,
- “user”: information about the user who triggered the artifact.

The script can read this file and use any of these properties for its processing. For example:

```
$raw_context = Get-Content -Raw -Path "$env:EVENTDATA"
```

```
$context = $raw_context | ConvertFrom-Json6
# Just address the parameters like this
"The artifact value is " + $context.artifact.value
"The incident name is " + $context.incident.name
"The kill_chain custom field is " + $context.incident.properties.kill_chain
"The user who initiated the action is " + $context.user.email
```

Results

By default, the results are attached as a text file to the incident.

Configuration

The default behavior above only requires that you

- configure the action, using the administration UI,
- deploy your script to the “scripts” folder of the repository.

This behavior can be changed, or extended for specific commands, using the integration config file.

The default “[shell]” section of the config file has:

```
# ---------------------------------------------------------------------------
# The 'shell' action component
# ---------------------------------------------------------------------------
[shell]
queue=shell
command=powershell "C:\\resilient\\Henosis\\app\\scripts\\{{action_name}}.ps1"
"{{artifact.value}}"
result_disposition=new_attachment,filename={{action_name}}_{{properties._messa
ge_headers.timestamp|iso8601}}.txt
```

This creates the default behavior described above:

- Listens to actions on the message destination (queue) named “shell”
- For any action, executes a PowerShell script named by the action name, passing the artifact value as a command- line parameter
- Creates a new attachment from the result, with filename consisting of the action name and timestamp.

Additionally you can create specialized version of these command and result-disposition settings for any specific action. For example, to have an action named “nslookup” directly run the nslookup command:

```
nslookup=nslookup "{{artifact.value}}"
nslookup_result_disposition=new_attachment,filename={{action_name}}_{{artifact
.value}}.txt
```

Alternative result dispositions may be useful for some actions. Dispositions include:

- update_incident – the result is interpreted as JSON, which is applied as an update to the fields of the incident that the action was initiated from.
- update_task – the result JSON updates the task where the action was applied.
- update_milestone, update_note, update_artifact: similarly.
- new_incident, new_task, new_note, new_milestone, new_artifact: similarly.

Other Considerations

Shell commands execute from a thread pool with 10 threads by default. It's OK to have long-running commands, but eventually they will block other actions. There is no built-in timeout or supervisor-killer if commands run too long.

Shell commands run as the user account that is running the integration. Be particularly careful to review your custom actions for inadvertent "command injection" risk.