Developer Guide for Nexpose Ticketing System

Contents

RAPID7

Last Revised: January 26, 17

Solution Summary

The goal of incident management is to restore normal service operation as quickly as possible following an incident, while minimizing impact to business operations and ensuring quality is maintained. The integration with Nexpose allows customers to create incident tickets based on vulnerabilities found across their systems. Using the Nexpose Ticketing Gem will allow users to implement their own Integration endpoint within the ticketing system.

Before You Begin

The integration is written in Ruby and therefore requires Ruby binaries to be installed on the host system. The following link shows the different options for installing Ruby in several platforms:

- https://www.ruby-lang.org/en/downloads/

Once installed, the next step would be to check out the source code from GitHub, located at the following address:

- https://github.com/rapid7/nexpose_ticketing

Custom Helper

Requirements

A complete ticketing service helper consists of the following files:

1. **Helper:** This Ruby file should implement the methods necessary to transform a CSV report into the appropriate format for a specific ticketing service. For example, JIRA expects a JSON object for the ticket format. This class may be extended from the provided BaseHelper class. The guide for each currently existing helper contains descriptions for the methods used. These implementation specific descriptions may help with development of a new helper.

2. **Configuration:** The configuration file defines implementation specific details such as the username, password and address needed by the helper to create a ticket. Please note that the 'helper_name', 'vendor' and 'product' fields are mandatory. The 'helper_name' setting should match the class name of the helper exactly e.g. 'JiraHelper', 'ServiceNowHelper' etc.

Helper Implementation

The files should be implemented as follows:

1) **Helper File: <service>_helper.rb:**

The class contained within this file must implement the following methods:

1. initialize: The constructor which takes the ticketing and service-specific options. A call to 'super' ensures that objects common to each mode, such as the logger and ticket metric objects, are initialized.

2. prepare_create_tickets: Takes the provided CSV data and returns a list of new ticket objects. Each ticket should contain a unique NXID so that it may be identified for ticket updates and closures.

3. create_ticket (tickets): Takes the output from 'prepare_create_tickets' and sends the ticket creation requests to the service endpoint.

4. prepare_update_tickets: Takes the provided CSV data and returns a list of both new tickets and ticket updates. This may require querying the ticket service for existing ticket IDs.

5. update_tickets: Takes the output from 'prepare_update_tickets' and sends the ticket creation and update requests to the service endpoint.

6. prepare_close_tickets: Takes the provided CSV data and returns a list of tickets which may be closed. This may require querying the ticket service for existing ticket IDs.

7. close_tickets: Takes the output from 'prepare_close_tickets' and sends the ticket closure requests to the service endpoint.

2) **Configuration File: <service>.config**

The configuration file is a YAML document which defines the service information required by the helper class. An example config may look like:

```
# (M) Helper class name.
:helper_name: <Service>Helper
# (M) Product name
:product: <Service>
# (M) Vendor
:vendor: <Vendor>


# Optional parameters, these are implementation specific.
:endpoint_url: https://url/rest/api/
:username: username
:password: password
:service_settings: projectname
```

The mandatory fields are used throughout the integration to dynamically load dependencies, for example. The service name must be consistent across filenames and configuration options to ensure classes can be loaded and instantiated.

Optional parameters are implementation-specific and are used in the service helper class.

Packaging the Helper

The developed files must be stored in specific locations:

- Helper file: All helper files must reside under the '/lib/nexpose_ticketing/helper' directory as they are dynamically loaded when the ticket service starts.
- Configuration file: The configuration file should reside within the '/lib/nexpose_ticketing/config' folder.

The aforementioned folders may be found under the following paths:

- Windows: C:\Ruby<version>\lib\ruby\gems\<version>\gems\nexpose_ticketing\lib\nexpose_ticketing\
- Linux: /var/lib/gems/<version>/gems/nexpose_ticketing/lib/nexpose_ticketing/

Your installation folder may differ, please refer to the Ruby documentation for the specific location.

Initial Run

Assuming you've properly configured the Nexpose and helper parameters, call the executable from the command-line using the service name as a parameter:

```
ruby nexpose_ticketing <service>
```

Every time this command is run the service will query Nexpose, obtain any new vulnerability information and pass it to the custom helper.

Troubleshooting

The most common errors when running the script are:

- Configuration errors (such as incorrect indentation or user details).
- Specifying a user without sufficient permissions to create tickets.
- Specifying a Nexpose user without permission to create reports.
- Not specifying a site or tag.
- Specifying both a tag and a site. The tag will take priority and the site will not be scanned.

We recommend reading the log file under the log folder in the Gem - ensure to log information at important points in your custom helper to enable troubleshooting during development.

If everything still fails, please send an email to support@rapid7.com with the ticketing_service.log attached and a description of the issue.

Custom Modes

A class within the ‘modes’ folder represents each ticketing mode. Each mode is extended from the BaseMode class that contains a number of common helper functions that extract and format data from CSV rows. This class also contains the method stubs that must be extended for a fully implemented mode.

The dependency on a specific mode helper is loaded dynamically via the BaseHelper class as part of its initialization. Custom helpers must call ‘super’ within the initialization method to ensure that the mode is loaded.

Common Methods

These methods are provided as helper methods which each derived mode class may use. These may be modified to change the formatting of the tickets.

Method	Description
`finalize_description`	Generates a final description string based on a description hash, limiting its length to match the maximum specified within the configuration file.
`get_vuln_info`	Formats the vulnerability data contained within a CSV row, including its solutions.
`get_header`	Creates a header for a section regarding a specific vulnerability.
`get_short_summary`	Formats the vulnerability data contained within a CSV row, truncating it, if necessary.
`get_solutions`	Formats the solution data for a vulnerability.
`get_discovery_info`	Formats the ‘first discovered’ and ‘last seen’ information for a vulnerability.
`get_references`	Formats the references for a vulnerability as a list, limiting the number according to the maximum specified within the configuration file.
`get_assets`	Formats the assets associated with a vulnerability into a list.
`get_field_info`	Returns the fields that identify a ticket into a format suitable for logging.
`method_missing`	Method which is triggered when a method call on the helper is not recognized. This typically occurs when a helper has failed to extend a method.

Mode-specific Methods

The derived class must override these methods:

Method	Description
`initialize`	Performs any required initialization logic. Should call 'super' which stores the options and retrieves a reference to the logger object.
`get_matching_fields`	Returns the fields which the existing ticketing helpers use to collate CSV rows into tickets.
`updates_supported?`	Returns 'true' if the ticketing mode supports updates to existing tickets, otherwise 'false'.
`get_title`	Returns the ticket title.
`get_nxid`	Generates an NXID (a unique identifier which is inserted into the ticket description) based on row data.
`get_description`	Gets the initial description hash that is built up over several rows.
`Update_description`	Updates the description hash with information from the next row.
`print_description`	Finalizes the ticket description and converts it to a String object.

Queries

Queries are defined in the queries.rb file and are selected on the basis of ticketing mode, taking into account whether there has been a pre-existing scan. When designing new queries, please use the existing queries as templates.

Some queries have variations specifically designed for a ticketing mode e.g. “all_new_vulns” has both “all_new_vulns_by_ip” and “all_new_vulns_by_vuln_id” versions for IP and Vulnerability mode, respectively.

There are several main queries used by the ticketing system:

Query	Description
`last_scans`	Gets all the latest scans IDs.
`all_new_vulns`	Gets all new vulnerabilities for all sites.
`new_vulns_since_scan`	Gets all new vulnerabilities happening after a reported scan id.
`old_vulns_since_scan`	Gets all old vulnerabilities happening after a reported scan id.
`all_vulns_since_scan`	Gets all vulnerabilities happening after a reported scan id. This result set also includes the baseline comparison ("Old", "New", or "Same") for ticket updates.
`old_tickets`	Returns the vulnerability and asset for data for any open tickets which may be closed.