Playbooks Guide

FortiSOAR 7.0.2

TABLE OF CONTENTS

Change Log

Date	Change Description
2021-10-01	Initial release of 7.0.2
2022-02-16	Updated the "How do I change the timeout limit for playbooks?" questions, which is part of the FAQs in the Debugging and Optimizing Playbooks chapter.

Introduction to Playbooks

Playbooks in FortiSOAR allow you to automate your security processes across external systems while respecting the business process required for your organization to function. Playbook templates can be customized to follow an organization's current procedures while leveraging the automation capabilities of FortiSOAR.

> Playbooks are the key to empowering your organization with the full benefits of orchestration for both the human and machine side.

Playbooks can leverage a number of different FortiSOAR capabilities, such as inserting new data records, sending email notifications, and even referencing specified conditions to determine what path(s) to continue executing. Playbooks are highly configurable and provide consistent and thorough execution of IR response plans, enabling swift triage and containment of any potential cybersecurity threats.

The Playbook Engine runs asynchronously, meaning as an independent service, within the FortiSOAR application. This allows for better scalability and also frees the Application Engine to focus on request execution for better responsiveness to human users.

Overview of Playbook Collections

Use Playbook Collections to organize your playbooks. A playbook collection is similar to a folder structure in which you create and store playbooks that can be used for a particular strategy in your environment.

We recommend the following organizational scheme for storing your playbooks in Collections.

- Each integration target should have its own Collection, e.g., Splunk
- Actions should have their own Collections, such as Forensics, Enrichment, and Remediation, and further, the actions can leverage the integration playbooks
- Response Plans should have their own Collection and should leverage the Actions in a sequence based on the standard categories of incidents

Overview of Playbooks

Playbooks are individual sequences of steps designed to accomplish a specific purpose. Playbooks are akin to a functional programming language, with capabilities to handle internal processes and external integrations.

> FortiSOAR supports RBAC for playbooks and therefore administrators require to assign roles with appropriate permissions to users who require to work with playbooks. For example, for users who require to run playbooks must be assigned the `Execute` permission on the `Playbooks` module.

The Playbook Designer supports **Pan** and **Zoom** tools. In case of large playbooks, you can use the Pan tool to scroll through your playbook, and you can use the Zoom tools to view the details of the playbook.

Playbooks are executed by default in the context of Playbook Appliance (PBA).

Ensure that when you are creating a playbook that you give the PBA all the necessary privileges on all the modules that will be consumed while executing the playbook. For example, if you want to extract indicators from an incident record using a playbook, then the playbook must have a minimum of **Read** permission on the `Incident` module and the **Create** permission on the `Indicator` module..

Permissions required to work with playbooks

- To create Playbooks; you must be assigned a role with a minimum of `Create`, `Read`, and `Update` permission on the `Playbooks` module.
- To modify steps and to view steps in-depth, you must be assigned a role with a minimum of `Read` and `Update` permission on the `Playbooks` module.
- To view the Playbook Designer (you cannot view the steps in detail), you must be assigned a role with a minimum of `Read` permission on the `Playbooks` module.
- To create and delete Playbooks, you must be assigned a role with a minimum of `Create`, `Read`, `Update`, and `Delete` permission on the `Playbooks` module.

Setting the logging levels for playbooks

From version 7.0.2 onwards, you can choose to set the logging levels for individual playbooks to either `INFO` (default) or `DEBUG`. Your administrator sets the global playbook logging levels; however, an administrator can also enable a setting that allows users to change the logging level for individual playbooks, which in turn overrides the global playbook logging levels. Set the playbook level logging to `INFO` for production instances and in scenarios where you want to use storage space efficiently; whereas use the `DEBUG` option only while designing or debugging playbooks since this option can quickly fill up the storage space.

To set the logging levels of a playbook, open the playbook in the playbook designer, and then click **INFO** or **DEBUG** (depending on the logging level set), which is present at the top of the Playbook Designer, as shown in the following image:
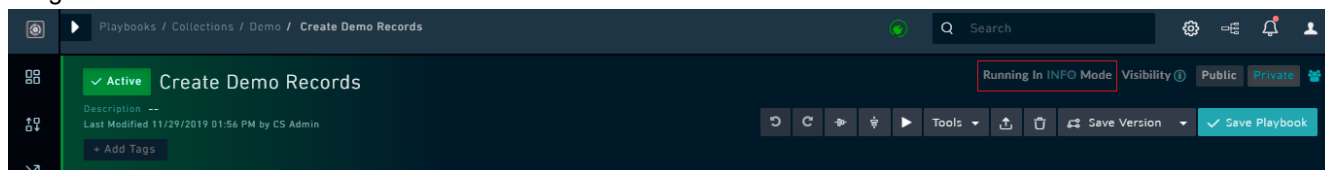
Clicking **INFO** or **DEBUG** displays the `Playbook Execution Log Level` dialog:
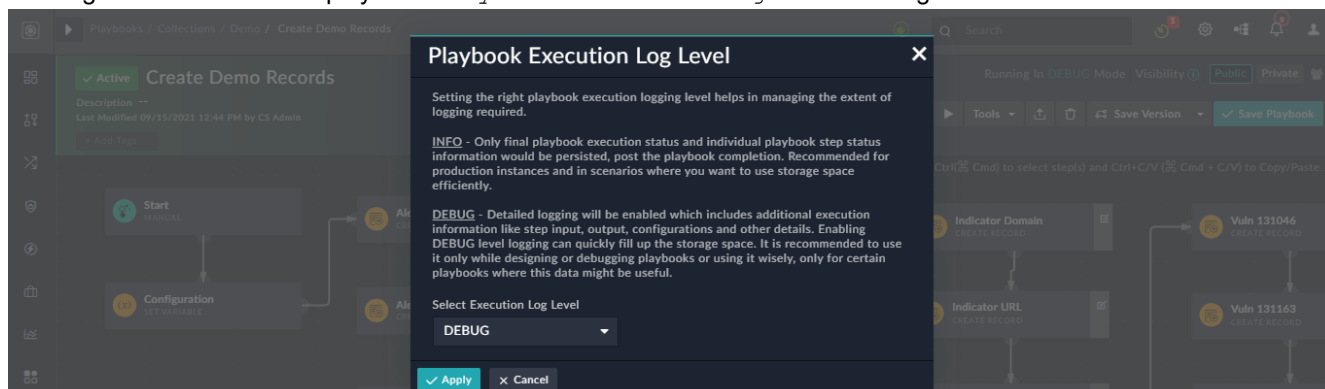
From the **Select Execution Log Level** field, select the logging level that you want to set for this playbook and click **Apply**. For example, if you select **DEBUG**, you will see that text at the top of the Playbook Designer changes to **Running in DEBUG mode**. Click **Save Playbook** to apply this change to the playbook.

For more information on playbook execution logs, see the Debugging and Optimizing Playbooks chapter.

Assigning ownership of playbooks

You can assign ownership to playbooks, i.e., if you want certain playbooks to be executed only by certain teams, then you can create a Private playbook and assign the playbook to only those teams.

By default, when you are creating a playbook, the playbook is created as a Public playbook, i.e., the playbook can be executed by all (if they have other appropriate rights). However, you can change this to Private by clicking the **Private** button that is present at the top of the Playbook Designer, as shown in the following image:

To assign the playbook to a particular team, click the **Teams** icon ( ). This opens the `Assign Owners` dialog. In the `Assign Owners` dialog, from the **Owners** drop-down list select the team that will own this playbook and click **Assign**.
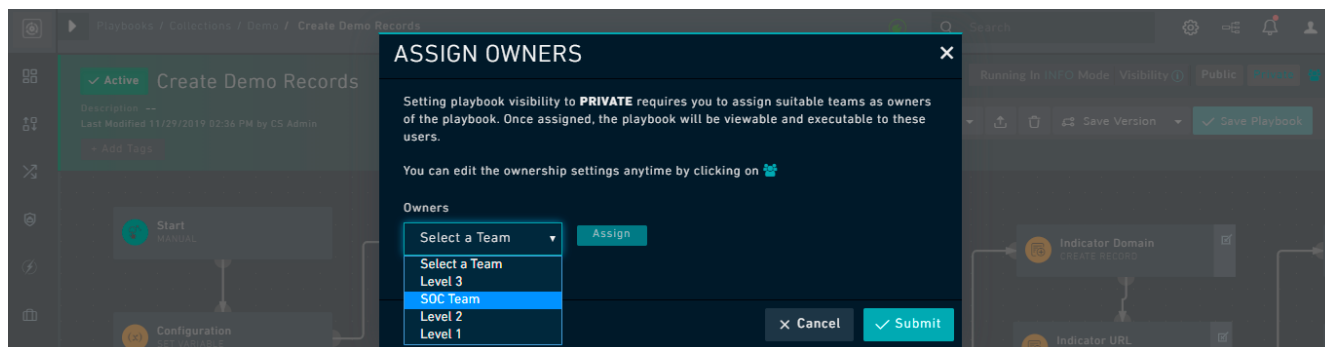
You can make multiple teams, owners of this playbook in a similar manner. If you want to remove ownership from a particular team, click the **red cross** that appears besides the team name.
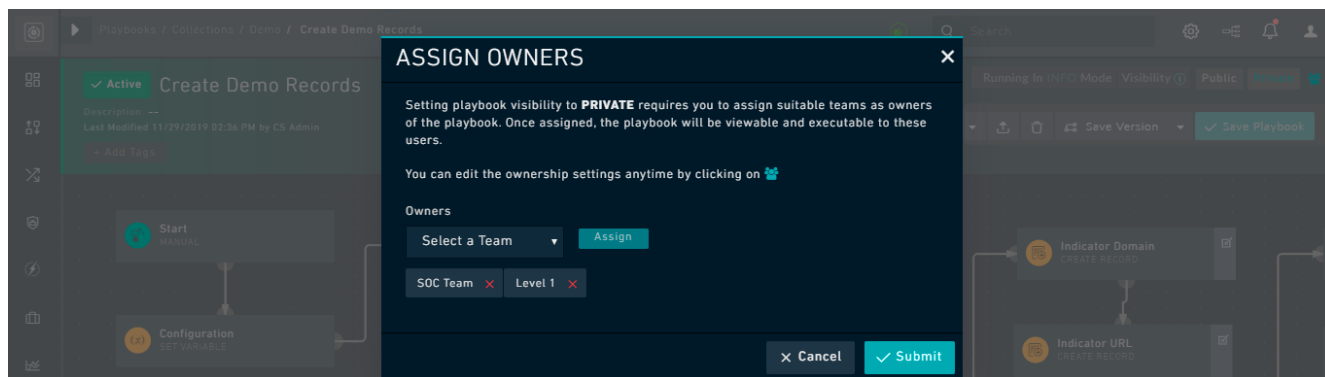
It is important to note that execute actions such as **Escalate**, **Resolve**, or any actions, which are displayed in the **Execute** drop-down list in records of modules such as Alerts, are shown based on ownership. For example, if you have created a **Private** playbook with a Manual Trigger or a Custom API Endpoint trigger on the `Alerts` module, and if you go to the alerts module and select the record, then **Execute** drop-down list will contain only those playbooks that belong to your team(s). In case of On Create or On Update triggers, RBAC is honored by matching the team defined in the playbook with the teams associated with the record.

When you export a playbook collection then all the playbooks within that collection become "Public" playbooks, even if some were marked as "Private playbooks, and the owners of the private playbooks become blank. Therefore, when you import these playbooks back into FortiSOAR, and you want the playbooks to be private, then open the playbook and click "Private" and reassign the owners. Exporting a single private playbook also marks it as public and its owners also become blank, and therefore, after importing this playbook into FortiSOAR, you will have to follow the same steps to make it "Private", if you want this playbook to be a "Private" playbook.

Creating Playbooks

1. Click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click **New Collection** to define a new playbook collection in which to save the playbook you want to create, or, click an existing playbook collection and add the new playbook in that collection.
   Note: You cannot add a playbook directly on the `Playbook Collections` page, you require to add playbooks to a playbook collection.
3. In the `Add New Playbook Collection` dialog, add the name of the collection in the **Name** field and optionally in the **Description** field, add the description for the playbook collection.
   You can optionally change the icon that represents the playbook collection, by clicking **Change Image** and dragging and dropping your icon to the Upload an Image dialog, or browsing to the icon on your system, selecting the icon and then clicking **Save Image**.
   You can optionally also add keywords in the **Tags** field that you can use to reference the playbook collection and making it easier to search and filter playbook collections and playbooks. You can add special characters and spaces in tags; however, the following special characters are not supported in tags: `'`, `,`, `"`, `#`, `?`, and `/`.
   Click **Create** to create the new playbook collection.

4. To add a playbook, click the collection in which you want to create the new playbook, and then click **Add Playbook**, which displays the `Add New Playbook` dialog:
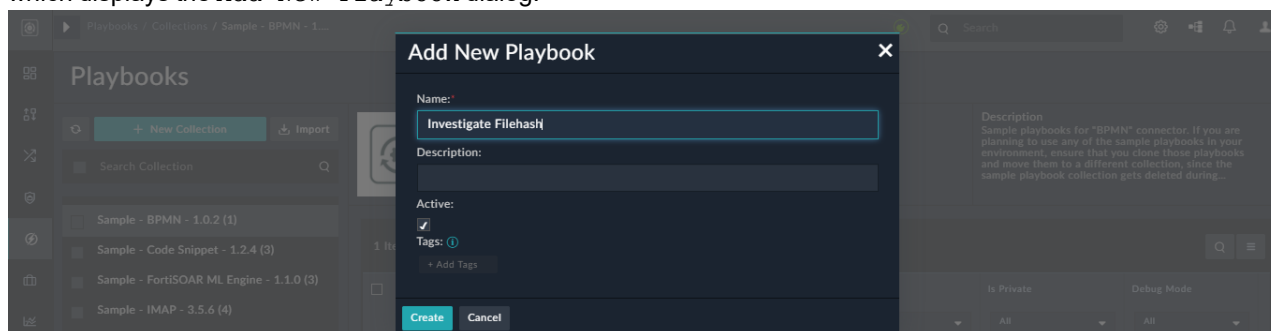
5. In the `Add New Playbook` dialog, add the name of the playbook in the **Name** field and optionally in the **Tags** field, add keywords that you can use to reference the playbook, making it easier to search and filter playbooks. You can optionally in the **Description** field, add the description for the playbook.
   Important: Playbook names must be unique within a collection.
   The **Active** checkbox sets the state of the playbook as `Active`, or `Inactive`. By default, the **Active** checkbox is selected, i.e., new playbooks are created in the `Active` state.
   Click **Create** to add the new playbook.
   Note: The logging level of new playbooks are set as `INFO`, if you want detailed logging for the playbook, then you can open the playbook and set its mode to `DEBUG`.

6. FortiSOAR displays the `Playbook Designer` for the newly added Playbook, with a placeholder trigger step and the name you have specified being displayed in the **Name** field at the top of the Designer. Now, you must select a playbook trigger from the `Triggers` section and enter the necessary variables for the selected trigger, and then click **Save**.
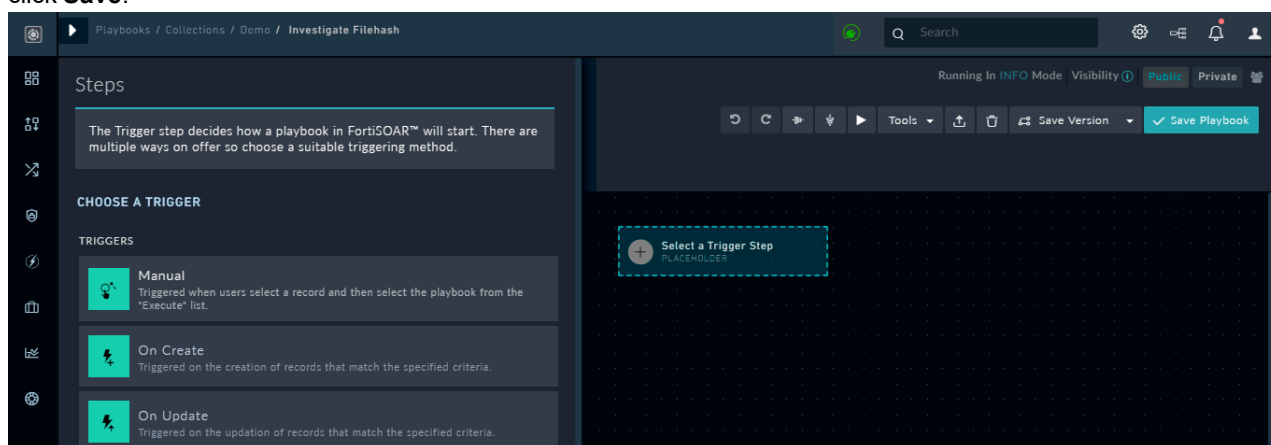
For information on the various triggers, see the Triggers & Steps chapter .
Note: Specific conditions that the playbook should meet before continuing can be called out by creating a `Decision Step` immediately after the trigger. While the playbook will still execute, the decision step (s) determines if the playbook continues through the following steps or is considered finished.

7. Add playbook steps.
   Once you've selected a trigger, FortiSOAR displays the trigger step in the `Playbook Designer` with highlighted connector points as shown in the following image:
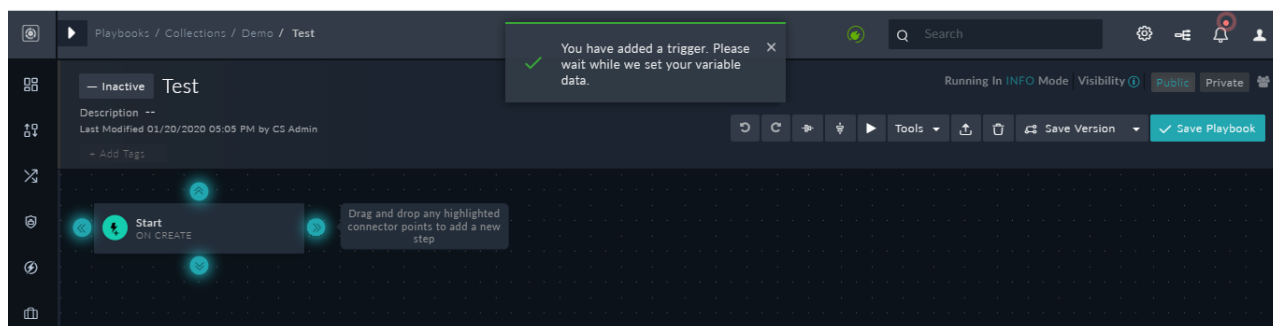
Drag-and-drop a connector point to connect to another playbook step. FortiSOAR adds a placeholder step on the playbook designer page and opens the `Steps` tab which displays all the available playbook steps, select the playbook step that you need next, add the **Step Name** and the required variables and click **Save**.
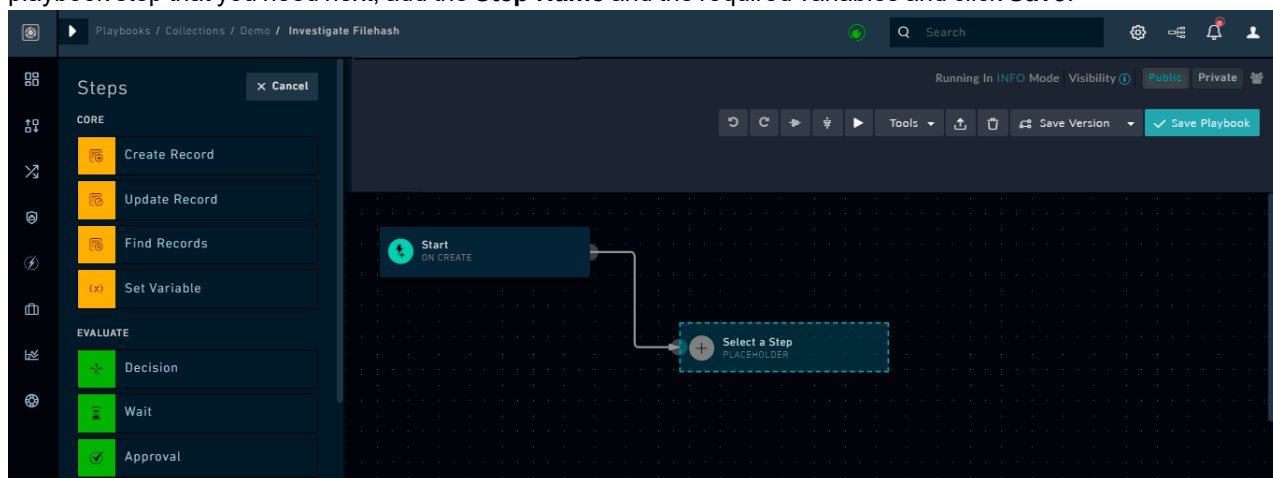
Similarly, you can add further steps and create the desired flow for the playbook. A playbook ends when there are no additional steps to run. For more information on steps, see the Triggers & Steps chapter.

8. Connect playbook steps or remove a connection between steps.
   It is straightforward to connect playbook steps as well as to remove the connection between playbook steps.
   To connect a playbook step, use the connection points that appear when you hover on a Playbook step. Select a connection point and drag and drop the arrow connector on the step you want to connect.
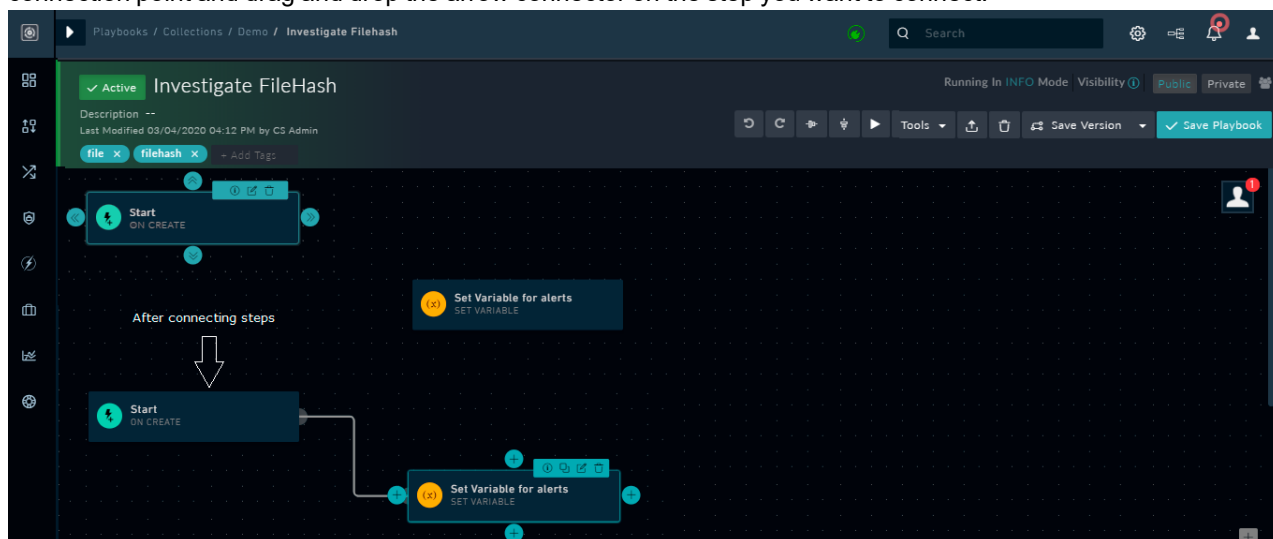
To remove a connection between playbook steps, hover on the arrow connector between the steps, which then

displays a cross (**X**) red color. Clicking **X** displays a `Confirm` dialog, click **OK** to remove the link between the playbook steps.
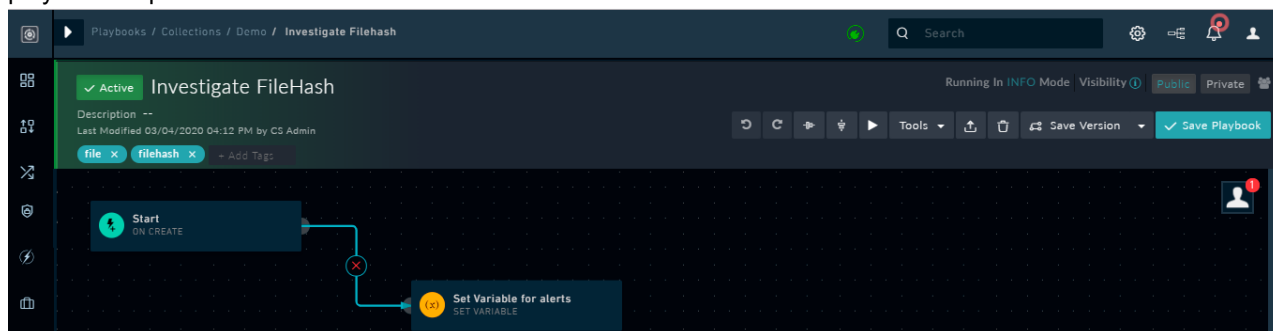
9. (Optional) To edit or remove an existing playbook step double-click on the step to reopen it and then you can edit the step or delete the step entirely by clicking **Delete Step**.
Playbook steps include icons for the **Info**, **Edit**, **Clone**, and **Delete** actions enabling you to perform these actions in the step itself using the respective icons.
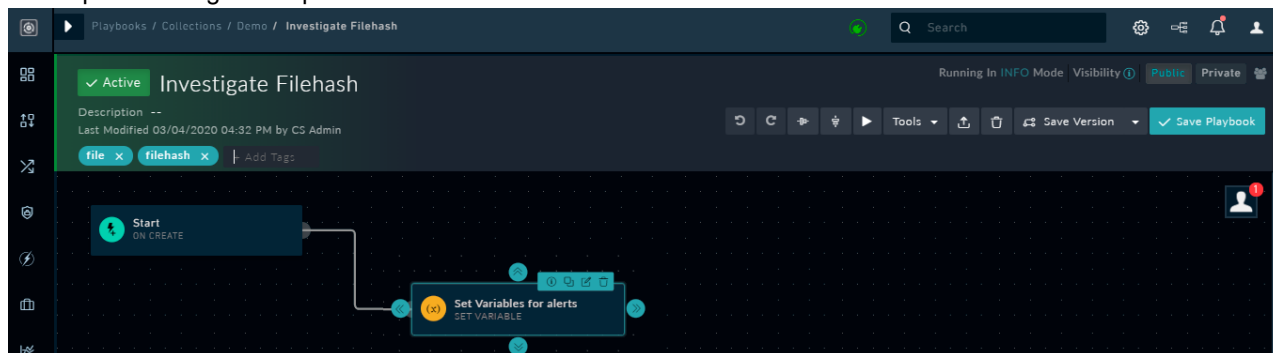
Clicking the **Info** icon displays additional information, if available, about the step.
Clicking the **Clone** icon creates a copy of the current step and opens the step with the name as `Copy of %Step Name%`. All the properties of the current step are copied to the cloned step. You can edit the properties of the cloned step as required and then save the step.
Clicking the **Edit** icon reopens the step, and you can edit the properties of the step and then save the step.
Clicking the **Delete** icon deletes the step entirely.

Importing the BPMN Shareable Workflows as FortiSOAR Playbooks

FortiSOAR provides you with the ability to convert a BPMN Shareable Workflows to FortiSOAR playbooks. Business Process Model and Notation (BPMN) is a tool using which you can create flowcharts, and these flowcharts tend to be specific towards cybersecurity workflows. Therefore, this feature provides you with the advantage of importing your BPMN workflows and directly converting them into FortiSOAR playbooks, without the need to again create the same workflow in FortiSOAR.

The feature is introduced as a "BETA" feature with more enhancements being planned to be added in the subsequent releases to make the BPMN import more robust.

Import the BPMN Shareable Workflows into FortiSOAR as follows:

1. Export your BPMN Shareable Workflows from your tool, such as Flowable or Camunda.
BPMN workflows are exported in the XML format.
2. To import the BPMN workflows into FortiSOAR:
Note: FortiSOAR supports importing only a single BPMN workflow, i.e., you cannot import a collection of BPMN

workflows.

a. Log into FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.

b. Click **Import BPMN**, which opens the `Import BPMN` dialog.
Note: We are providing a "BETA" Version of this feature so that users can get a preview of this feature.

c. In the **Import BPMN** dialog, do the following:

  i. From the **BPMN Tool** drop-down list, select the tool in which you have created your BPMN workflows.
Note: FortiSOAR supports Flowable and Camuda.

  ii. From the **BPMN Output Format** drop-down list, select the output format in which you want to convert your BPMN workflow.
Note: FortiSOAR supports only XML as an output format.

  iii. Drag and drop the BPMN XML file, or click the **Import** icon and browse to the XML file to import the BPMN XML file into FortiSOAR.
If the XML of the BPMN workflow does contain errors, then a warning will be displayed in the `Import BPMN` dialog, which will contain the reason why the XML cannot be imported into FortiSOAR.
If the XML of the BPMN workflow does not contain any mismatched elements or any other errors, then you will be able to import the workflow as a playbook in FortiSOAR.
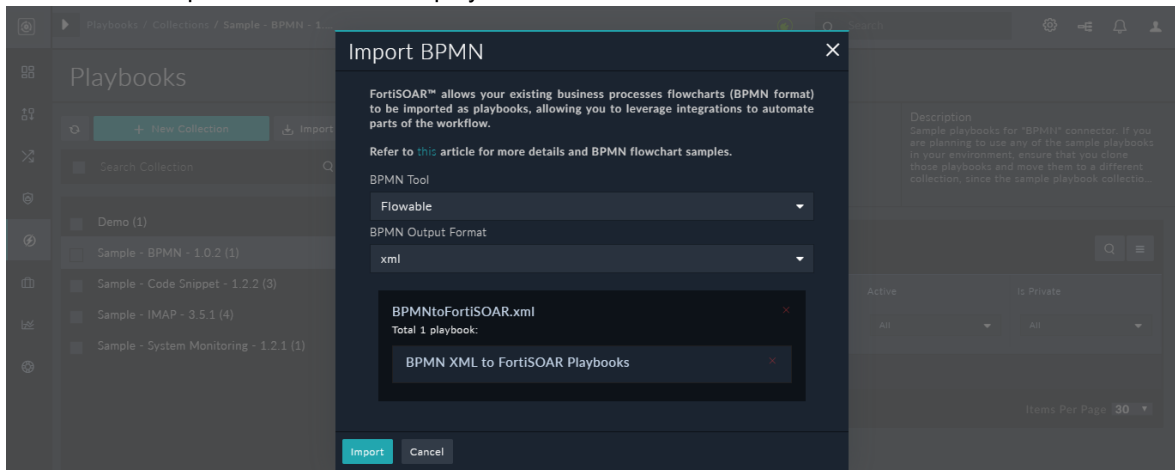
  iv. To import the BPMN workflow file, click **Import**.
This imports the workflow as a playbook in FortiSOAR with the same name as the workflow.
Note: The name of the playbook must be unique, i.e., if you have two workflows with the same name that you want to import, you must either change the name of the playbook or click the **Replace existing playbook** checkbox to replace the existing playbook.
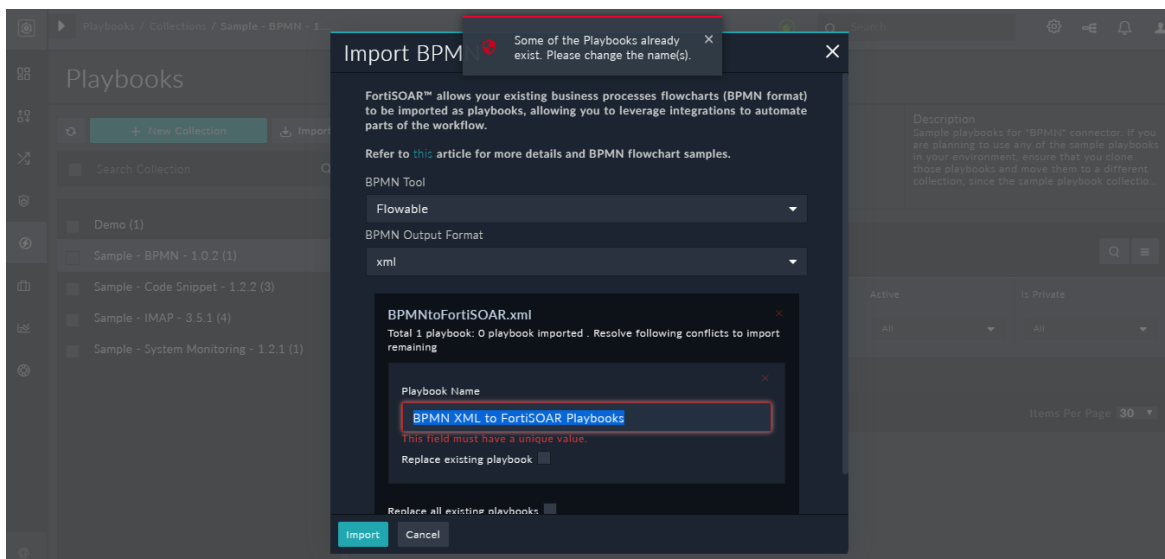
FortiSOAR displays the imported workflow in the Playbook Designer as shown in the following image:
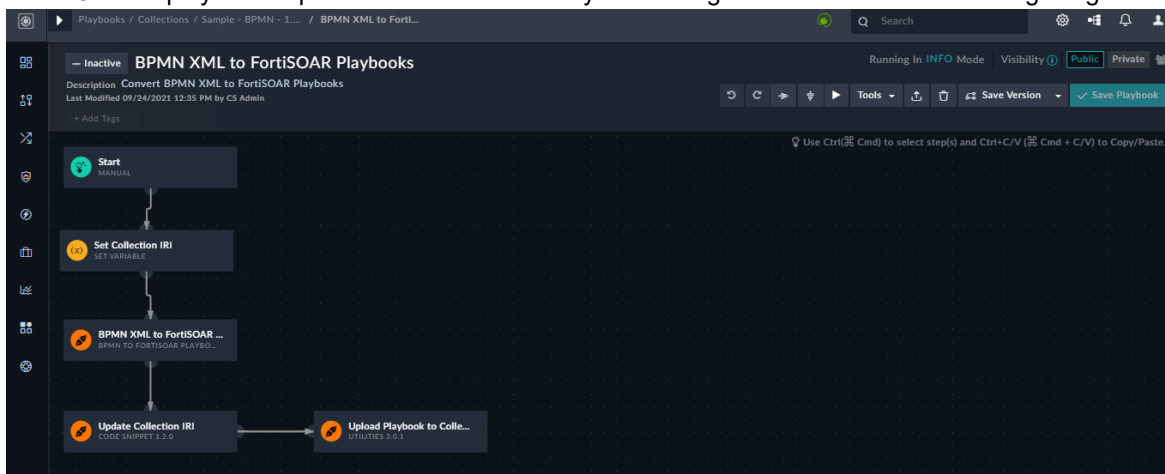
Now you can edit the playbook as required in the playbook in FortiSOAR and easily create the automated workflow.

Translation of BPMN workflow steps into FortiSOAR steps in playbooks

The following table specifies which BPMN (*Flowable* in this case) workflow steps maps to which of the FortiSOAR steps in the playbooks:

Flowable (BPMN) step	FortiSOAR steps	Notes
SequenceFlows	Routes	Any `SequenceFlows` defined in your BPMN workflow get converted to a `Decision` step in FortiSOAR playbooks.
StartEvents	Trigger steps	Your BPMN workflow must mandatory have a "Start" event which is the starting point of the BPMN workflow. The Start event in the BPMN workflow get converted to a `Manual Trigger` in FortiSOAR playbooks.

Gateways	Decision Step	Your BPMN workflow must mandatorily have a "Flow Condition" input which must be referenced to the Gateway ID.
UserTasks	Manual Tasks step	Note: If the `<userTask>` is not created according to FortiSOAR Manual Task step requirements, then a generic manual task step is created in the FortiSOAR playbook instead of failing the playbook. After you import the workflow you can update the manual task step.
ServiceTasks	Create Record step Or Update Record step	A `<serviceTask>` in your BPMN workflow must have the following: - A "Class" attribute to validate the model. - The "Class" attribute must be specified as a `module` - Addition of a "Class field" which contains either Create or Update.
ScriptTasks	Connector step or as a Code Snippet step	A `<scriptTask>` in your BPMN workflow must have the following: - Name = `{{ConnectorName}}` - scriptFormat = `{{FortiSOAR Connector Action}}` - `<script>` => CDATA[`{{property mapping}}`] **Note**: If the connector that you have defined in the `<scriptTask>` step is not installed in your FortiSOAR instance, then a generic connector step is created in the FortiSOAR playbook instead of failing the playbook. After you import the workflow you can update the connector step.

MailTasks	SMTP step	The `mailTask` is type of a `<serviceTask>` and it must be defined in your BPMN workflow as following:
		`<serviceTask>`
		`Flowable:type = mail`

HttpTasks	FortiSOAR Utility Step (REST API call)	The `httpTask` is type of a `<serviceTask>` and it must be defined in your BPMN workflow as following:
		`<serviceTask>`
		`Flowable:type = http`

Working with Playbooks

1. Click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, you can search, import, export, or delete a playbook collection.
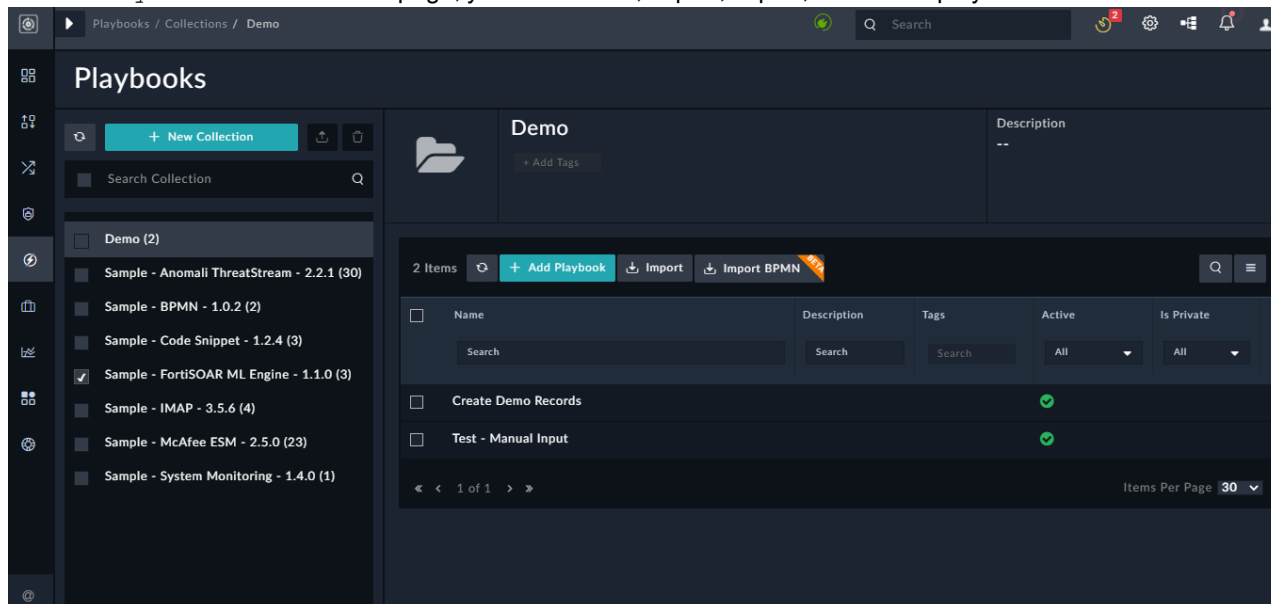
Use the **Search Collection** field to search for playbook collections.

You can import a playbook collection into FortiSOAR if it is in the appropriate JSON. To import a playbook collection into FortiSOAR, on the `Playbook Collections` page, click **Import**.

On the `Import Collections` dialog, drag and drop the JSON file, or click the **Import** icon and browse to the JSON file to import the playbook collection into FortiSOAR and then click **Import**.

Note: The name of the playbook collection being imported must be unique.

If you want to replace an existing playbook collection, then you must click the **Replace existing playbook collection** checkbox.

FortiSOAR also displays the list of macros that would be imported along the playbook collections or playbooks on the `Import Playbook` dialog. These are the macros that were part of the playbook that you had exported. You can review the imported macros and choose to modify them as per your requirements.

If the JSON format is incorrect, FortiSOAR displays an error message and does not import the file.

If the JSON format is correct, FortiSOAR imports the playbook collection and displays a success message.

Note: Any tags associated with the playbook collection are upserted into the system when you import a playbook collection.

To export a playbook collection, select the playbook collection and click **Export**. Any tags associated with a playbook collection are exported when you export a playbook collection.

FortiSOAR exports the playbook collection in the JSON format.

To delete a playbook collection, select the playbook collection and click **Delete**. Users with `Delete` permissions on the `Playbooks` module can delete playbook collections.

FortiSOAR displays a confirmation dialog, and once you click **OK**, the playbook collection is deleted.

You can click the **More Options** icon (⊞) to export records from the playbooks listing view in the csv or pdf format.
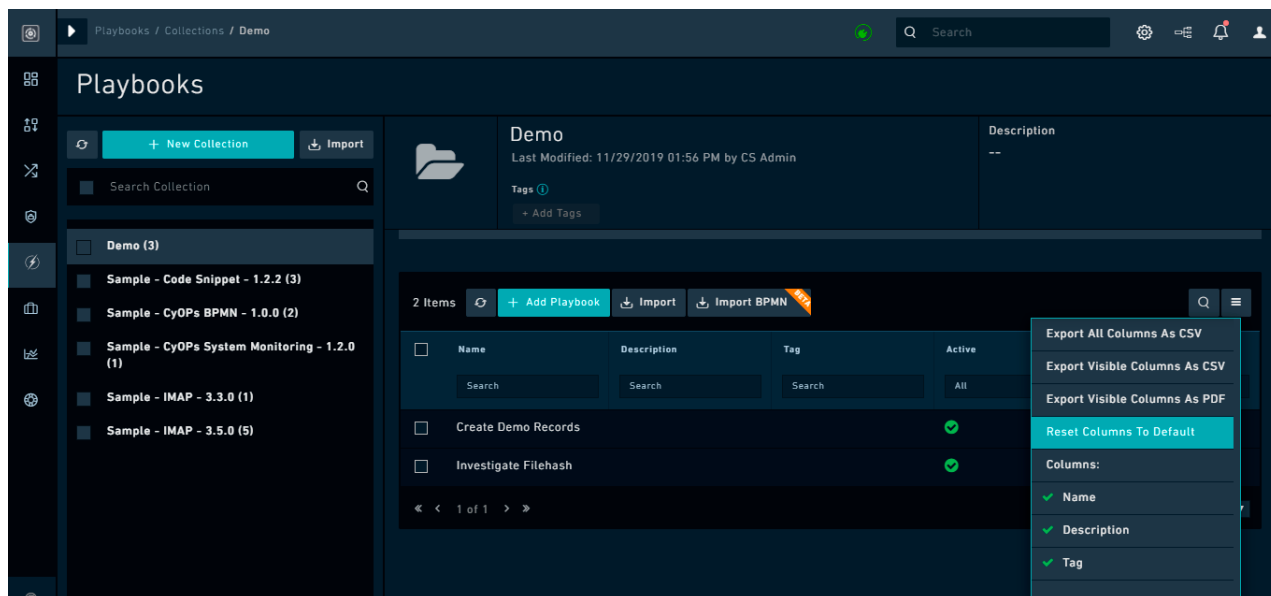
You can also reset the playbook record fields to the default fields specified for the playbook module, click the **Reset Columns To Default** option. You can include the **Created By**, **Created On**, **Modified On**, and **Modified By** fields in a playbook records for tracking purposes.

To import a playbook on the `<Playbooks Listing>` page, click **Import Playbook**. The playbook must be in an appropriate JSON format. Any tags associated with the playbook are upserted into the system when you import a playbook.

3. To perform operations on playbooks, click the playbook collection and open the `<Playbooks Listing>` page and then select playbooks. On the `<Playbooks Listing>` page, you can perform the following operations: **Activate**, **Deactivate**, **Clone**, **Move**, **Export**, **Change Logging Level**, and **Delete**.
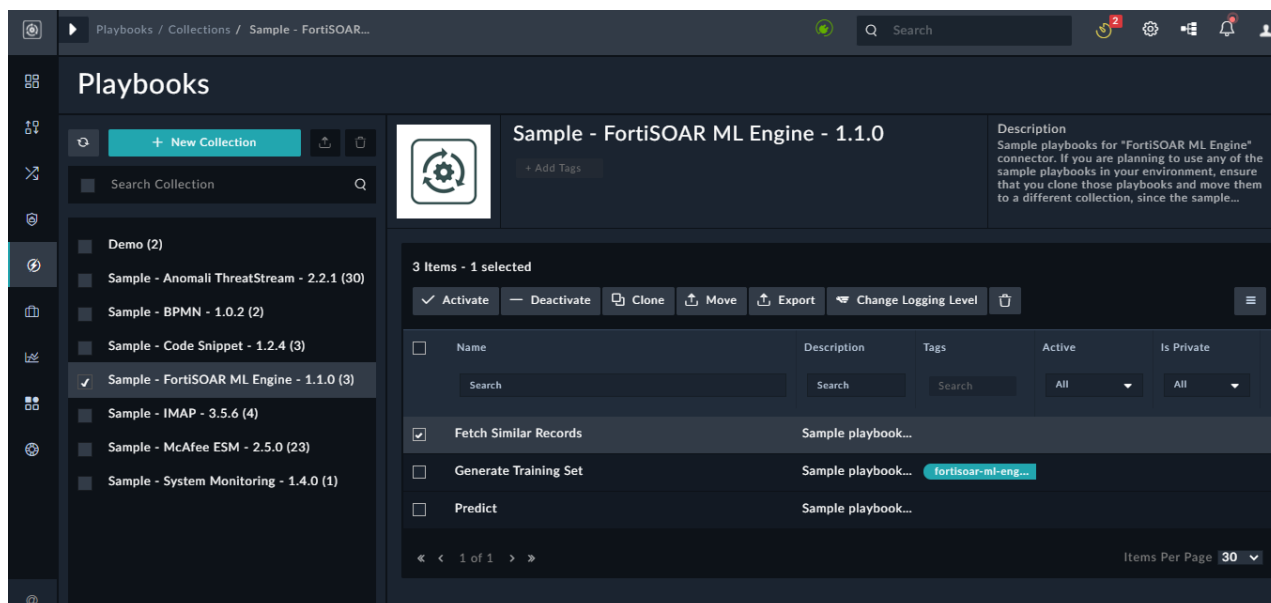
On the `<Playbooks Listing>` page, you might see a message such as `The count that you see on the playbook collection and the playbooks that you see.....` as shown in the above image. This message is shown since RBAC is enforced on playbooks, and this means that you can only see a listing of those playbooks for which you (your team) are the owner. As shown in the above image, The Demo collection shows that 3 playbooks are part of the collection. However, the `<Playbooks Listing>` page only displays a single playbook,

which means that the other playbook is a Private playbook with is owned by a team to which you are not assigned. You can also search for a playbook by typing keywords in the **Search** textbox.

Follow the same process as specified for the Playbook Collection import and the same restrictions as applies to the Playbook Collection import applies to the Playbook import.

To activate playbooks, select playbooks on the `<Playbooks Listing>` page and click **Activate**. To deactivate playbooks, select playbooks on the `<Playbooks Listing>` page and click **Deactivate**.

To clone playbooks, select playbooks on the `<Playbooks Listing>` page and click **Clone**. You might clone playbooks if you want to reuse the playbook as a starting point for a new playbook. Cloning the playbook clones every step within the playbook. You can select more than one playbook to clone at a single time. FortiSOAR clones the playbook and places the cloned playbook with the name as `Copy of %Playbook Name% (%New UUID%)`. Once you clone a playbook, you can edit it as per your requirements.

To move playbooks to another existing collection, select playbooks on the `<Playbooks Listing>` page and click **Move**. FortiSOAR displays the `Move Playbook` dialog that contains the `Move to collection` section. Clicking **Select** in the `Move to collection` section displays the `Collection` dialog. From the `Collection` dialog, select the collection to which you want to move the playbooks and click **Submit**.

To export playbooks, select playbooks on the `<Playbooks Listing>` page and click **Export**. FortiSOAR exports playbooks in the JSON format. Any tags associated with playbooks are exported when you export a playbook.

To change the logging level for playbooks, select playbooks on the `<Playbooks Listing>` page and click **Change Logging Levels**, which displays the `Playbook Execution Log Level` dialog. From the **Select Execution Log Level** field, select `DEBUG` or `INFO` as the logging levels for the playbooks and click **Apply**. For more information on playbook logging levels, see the Setting the logging levels for playbooks topic.

To delete playbooks, select playbooks and click **Delete**. Users with `Delete` permissions on the `Playbooks` module can delete playbooks.

4. To edit a playbook, on the `<Playbooks Listing>` page, click the playbook that you want to edit.

In the `Playbook Designer`, you can configure the following for the playbook:

Change the state of the playbook by clicking the **Is Active** box, for example, change the state of the playbook from `Active` to `Inactive`.

Change the **Name** of the playbook, by clicking the name box and updating the name. You can also add or update the **Description** of the playbook, or add or remove **Tags** from the playbook.

Modify the trigger for the playbook, change or add steps or actions to the playbook.

Use the **Tools** menu to enhance your playbook:
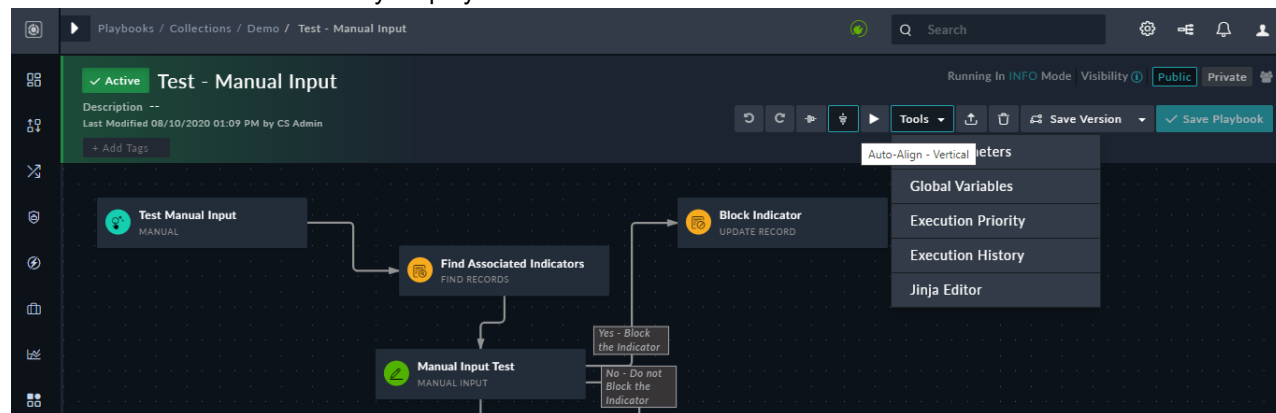
To add parameters, use the **Edit Parameters** option.

To add global variables, use the **Global Variables** option.

To view the execution history of the playbook, use the **Execution History** option. For more information see the Debugging and Optimizing Playbooks chapter.

To change the execution priority for a playbook, use the **Execution Priority** option. For more details, see the Changing the prioritization of playbook execution section.

To apply a jinja template to a JSON input and then render the output, use the **Jinja Editor** option. You can thereby check the validity of the jinja and the output before you add the jinja to the playbook. For more information, see the Dynamic Values chapter.

5. (Optional) Other actions that you can perform in the playbook designer are:
Use the **Export** button to export the playbook in the JSON format. Use the **Delete** button to delete the playbook.
Use the **Trigger Playbook With Sample Data** button to trigger the playbook from the playbook designer. For more details, see the Playbook Debugging - Triggering and testing playbooks from the Designer section.
Use the **Auto-Align - Vertical** and **Auto-Align - Horizontal** buttons to align the playbook vertically or horizontally. Version 7.0.0 introduces the Undo/Redo feature, which is very useful while building a playbook when there is a lot of trial and back and forth to be done. Use the **Undo** button or use `Ctrl+z`(Windows)/`Cmd+z` (Mac) to reverse changes made in a playbook and use the **Redo** button or use `Ctrl+y`(Windows)/`Cmd+shift+z`(Mac) to reverse the steps that you have undone; therefore, you can use the **Redo** operation only after you have performed the **Undo** operation in a playbook. The playbook designer displays messages about the effect of the Undo/Redo operation in the bottom-right corner. When you perform bulk operations such as moving, cloning, or deleting a number of steps in one go, clicking **Undo** reverts the step modification. Similarly, if you have modified a step and saved it, clicking **Undo**, reverts the step modifications. Note that when editing inputs in the step argument form, the browser's default change tracking is in effect; therefore, the Undo/Redo operations are applicable only after you save the step. Also, note that if you have made multiple changes in a small time period (around a second), then all these small changes are considered as a single operation.

6. Once you have completed updating the playbook, click **Save Playbook**.

Tips for working in the playbook designer

Following are some tips that you can use to make it easier for you to work with playbooks and playbook steps in the playbook designer:

- You can select a step by the `CTRL+Mouse click` operation. To select all the steps, press `CTRL+A`.
- You can drag and drop multiple selected steps.
- You can copy multiple selected steps by pressing `CTRL+C` or copy all the steps by pressing `CTRL+A` and then pressing `CTRL+C`. Ensure that you have clicked on your playbook canvas to bring it in focus before you copy the step(s).
  Note: The trigger step will not be copied.
- You can paste the copied step(s) into a different playbook by using `CTRL+V`. Ensure that you have clicked on your playbook canvas to bring it in focus before you paste the step(s).
  Note: You can also select **Paste** from the **Edit** menu in your browser to paste the copied steps.
- You can delete a step or multiple steps by selecting steps and pressing the `backspace` or the `delete` button.
- You can use the **Auto-Align - Vertical** and **Auto-Align - Horizontal** buttons to align the playbook vertically or horizontally. You can use these buttons to make your playbook look neat and organized, which is especially useful for very large playbooks where playbook readability might be an issue.

Playbook Debugging - Triggering and testing playbooks from the Designer

You can trigger playbooks directly from the playbook designer making it easier for playbook developers to test and debug playbooks while building them. Now, playbook developers do not require to go now to the module, select the record, and then choose playbook and then trigger the playbook and then come back again to the playbook designer to make the changes; all this can now be directly done from the playbook designer.

 Triggering a playbook from the designer starts the execution of the playbook, which can cause changes to your data leading to unwanted changes or loss of data. Therefore, it is important to review the playbook before it is triggered.

To trigger a playbook from the playbook designer, click the **Trigger Playbook with Sample Data** button. You can choose whether you want to use the **Last Run Data** as the sample data to trigger the playbook or you want to use **Custom Sample Data**.
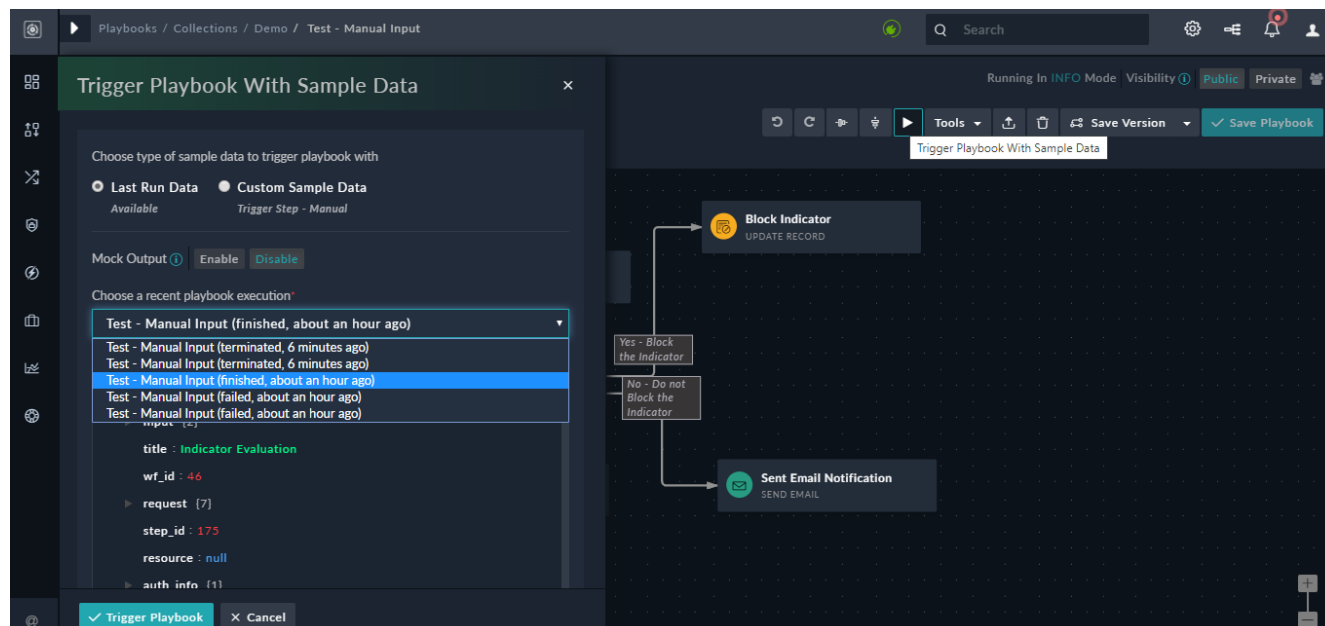
If you have run the playbook earlier, you can choose the **Last Run Data** option, and then from the **Choose a recent playbook execution** drop-down list, select the playbook execution with whose environment you want to trigger the playbook and click **Trigger Playbook**. Once you trigger the playbook with sample data, the `Executed Playbook Logs` dialog opens and you can view the logs and results of your executed playbook and continue to test and build your playbook.

You can also choose the **Custom Sample Data** option, and if you have a playbook that has a Manual trigger, then from the **Select Record** drop-down list choose the record(s) using whose data, i.e., fields and values, you want to use to trigger the playbook. Note that the 30 recently-created records will be fetched.
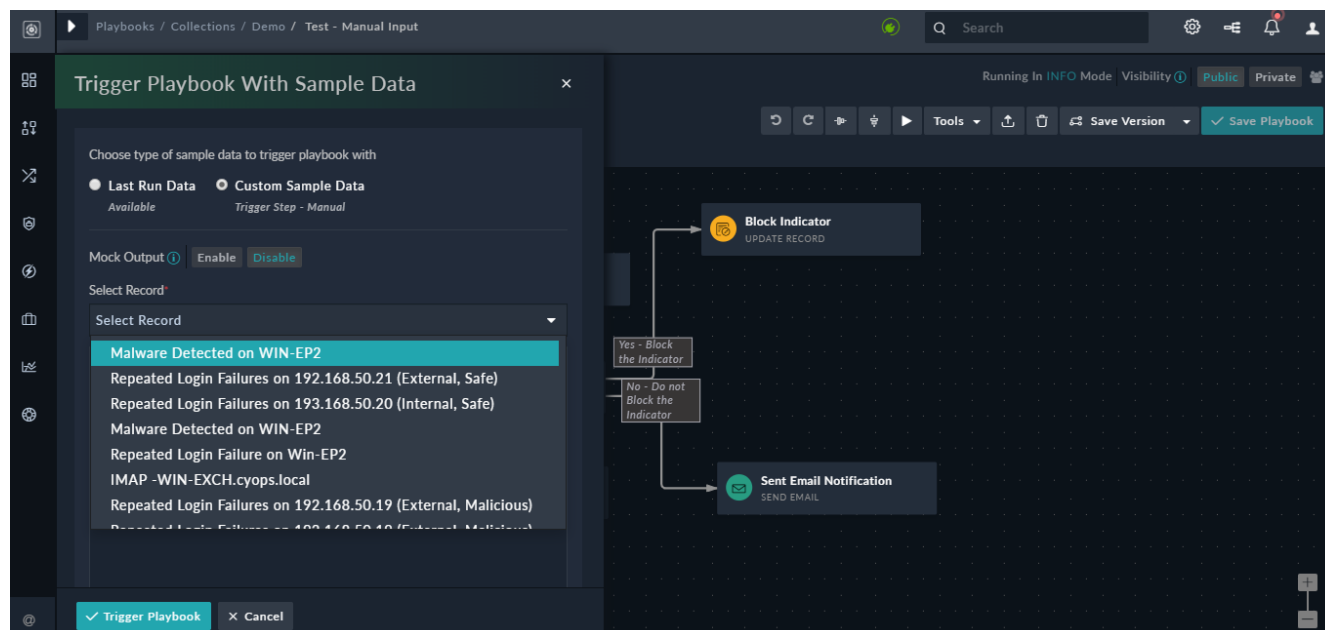
To trigger a playbook, you provide input based on the type of trigger you have defined for the playbook. For example, the **Select Record** drop-down list will not be present in case of a "Manual Trigger" step that has the **Does not require a record input to run** option selected since in this case the playbook does not require the data of a record to trigger a playbook. Also, in the case of a "Manual Trigger" step that has the **Run separately for each selected records** option selected, and in which you have selected multiple records and triggered a playbook from the designer, you will observe that only a single playbook will be triggered on a single record to simulate the output. Similarly, in case of a **Referenced** trigger, you can provide parameter values and trigger the playbook using those parameters.

The playbook can also use the "Mock Output" defined in the steps while running the playbook if you choose to **Enable** mock output.

Changing the prioritization of playbook execution

You can change the prioritization of playbook execution based on the importance of that playbook, thereby enabling the higher priority playbooks to be executed first even if there are some normal priority playbooks already queued for execution. Earlier, the playbook execution queue was based on first in first out method, with round robin assignment of workers (processes), which meant that important playbooks might get queued after lower-priority playbooks.

For example, if you have set up data ingestion to run every minute, then possibly you would have many data ingestion playbooks queued up, and then if you also require to run an important playbook with a manual action, it would earlier be run only once the data ingestion task that was scheduled before it was completed. Now, you can change the prioritization of the manual input playbook to "High" enabling it to get executed on priority.

You can set the priority for playbook execution as High, Medium, or Low. The default priority is set as "Medium". Playbook execution prioritization works as follows:

- If any worker is available for the task execution, it gets assigned a task from the "High" queue first and so on.
- If all workers are occupied with lower priority tasks and any higher priority task comes up, the high priority task gets executed only when any worker is again available.
- Low priority tasks do not get executed if there are high priority tasks.

To set a priority for playbook prioritization, open that playbook in the playbook designer. Click **Tools** > **Execution Priority**. In the `Execution Priority` dialog, you can set the playbook execution prioritization to **High**, **Medium**, or **Low**:
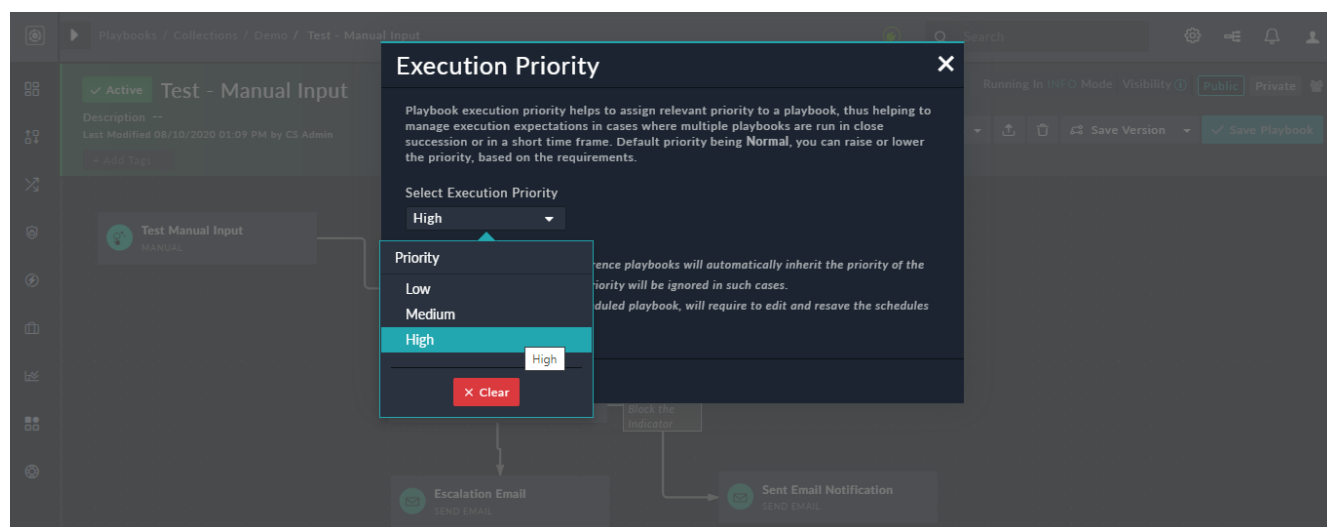
To list the number of messages (workflow count) in the '`celery`' queue, use the following command:
```
rabbitmqctl list_queues -p fsr-cluster --no-table-headers --silent | grep -E
```

```
"^\s*celery\s+" | awk '{print $2}'
```

When there is no queue, it will display 0 (default), and when the queue builds up, it will display the queue count number such as 10, 25, etc.

Notes:

- All 'sync' reference playbooks automatically inherit the priority of their parent playbooks, thereby ignoring any preset priority.
- If you update the execution priority of a scheduled playbook, then you require to edit and resave the schedules associated with that playbook.
- If you want to schedule a data ingestion playbook, then you must set the priority of the data ingestion playbook before scheduling the same.

FortiSOAR also integrates with a GUI-based celery monitoring tool called **Flower**, using which you can monitor and administer celery cluster and playbook execution queues. You can start a Flower web server using the following process:

```
cd /opt/cyops-workflow/sealab
```

```
../.env/bin/flower -A sealab --port=5555
```

Note: Ensure that port that you are specifying in the URL, 5555 in the above sample, is opened in your firewall and can be accessed.

Live User implementation in Playbook Designer

The playbook designer implements Live Users, which means that the playbook designer displays users who are also currently working on the same playbook. Therefore, when you open a playbook and if there are other users who are working on the same playbook apart from you, then the playbook designer will display the users working on the playbook, as well as the number of sessions that are active for each user. Live Users also notifies users that are working on the same playbook, if any other user or session has saved modifications to the playbook, so that the user can refresh the playbook before working on the same, thereby ensuring that users work on the latest version of the playbook. Users can also save versions of their current modified state of the playbook, thereby providing users with the ability to merge their changes.
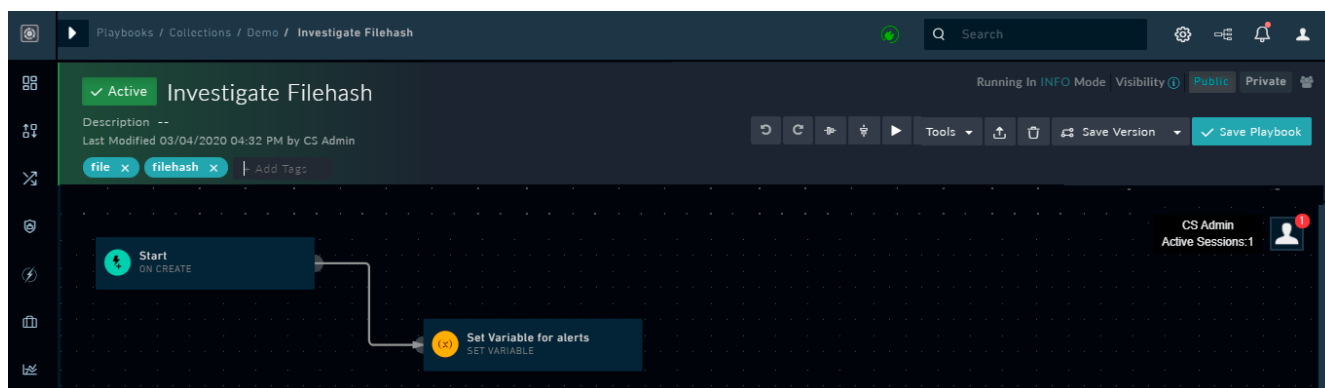
Live Users has the following benefits:

- Users are notified of other users or sessions that are active on the same playbook.
- Users work on the latest version of the playbook, and they do not lose their updates made to the playbook.

Saving versions of your playbook

You can save versions of a playbook that you are creating or updating. Using versioning, you can save multiple versions of the same playbook. You can also revert your current playbook to a particular version, making working in playbooks more effective.

The maximum number of versions that can be taken, across all users working on a playbook is 20. If you or other users try to take more than 20 snapshots, a confirm dialog is displayed that prompts you to delete a version so that you can free space and save a new version, as shown in the following image:
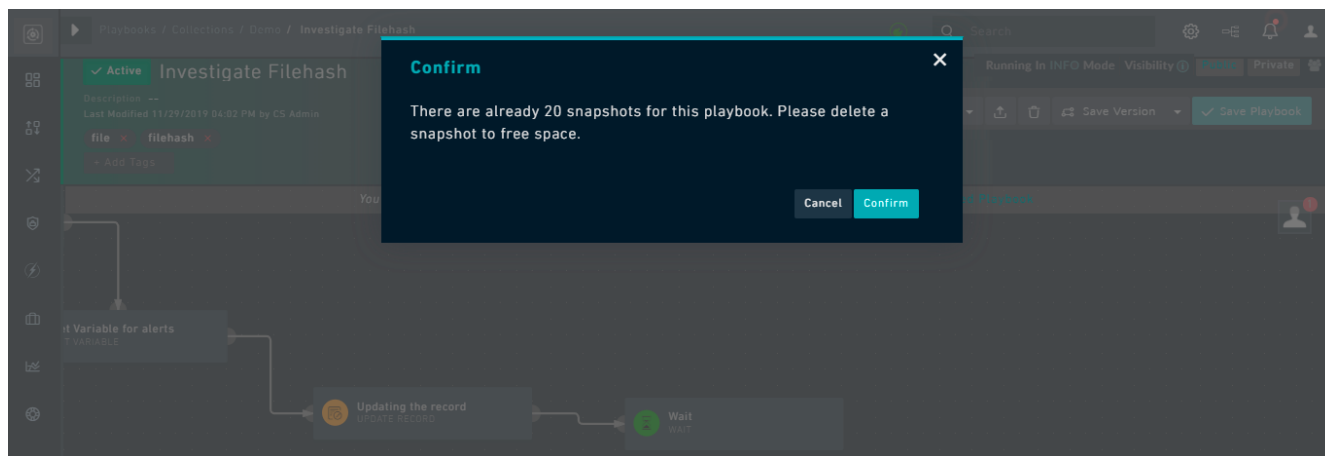
When you click **Confirm**, the `Versions - <Name of Playbook>` dialog is displayed. You can now choose the version(s) that you want to delete, click the **Delete** icon, and then click **Confirm** on the confirmation dialog and close the `Versions - <Name of Playbook>` dialog. This frees up space and you can now save a new version.

Using the `Versions - <Name of Playbook>` dialog, you can search for versions based on the notes you have added, and also filter versions by the Created By (user who has created the version), Saved On (time the version was saved) and action performed.

To take a snapshot and revert the playbook to a particular snapshot do the following:

1. In the playbook in which you are working in the playbook designer, click **Save Version**.

2. In the `Save Version` dialog, add a note that you want to associate with the version and click **Save Version**.
   It is recommended that you add meaningful notes for versions as these names will help you in identifying the snapshots when you want to revert to a particular version.

3. To revert a version, click **Save Version** and then either click **Revert to Last Saved** or click **View Saved Versions**.
   Clicking **Revert to Last Saved** reverts the playbook to the last saved version of the playbook
   Clicking **View Saved Versions** displays the `Versions - <Name of Playbook>` dialog that allows you to choose the version of the playbook to which you want revert:
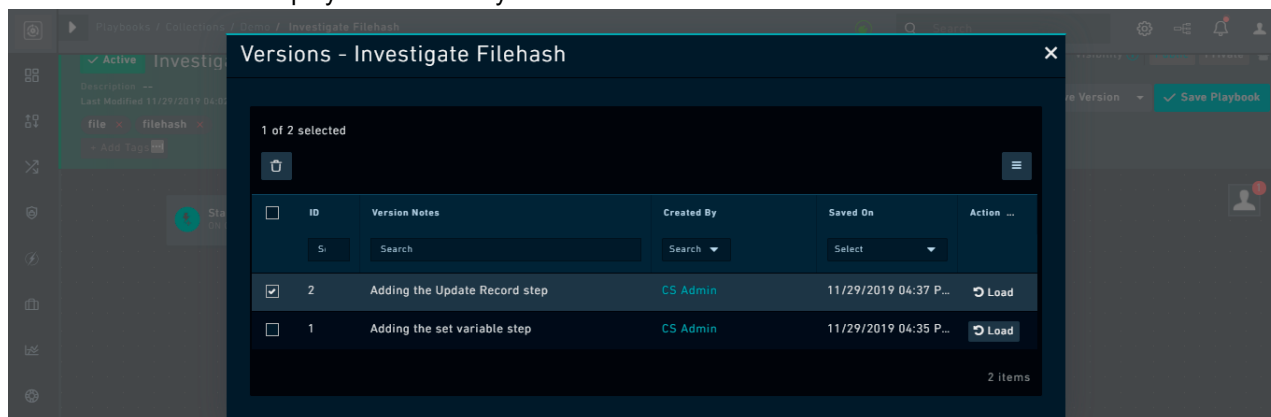
   In the `Versions - <Name of Playbook>` dialog, in the version row to which you want to revert, click **Load**.
   Once you click **Load**, that snapshot is loaded in the playbook designer, with a message: "You are working on a previously taken playbook snapshot...." as shown in the following image:
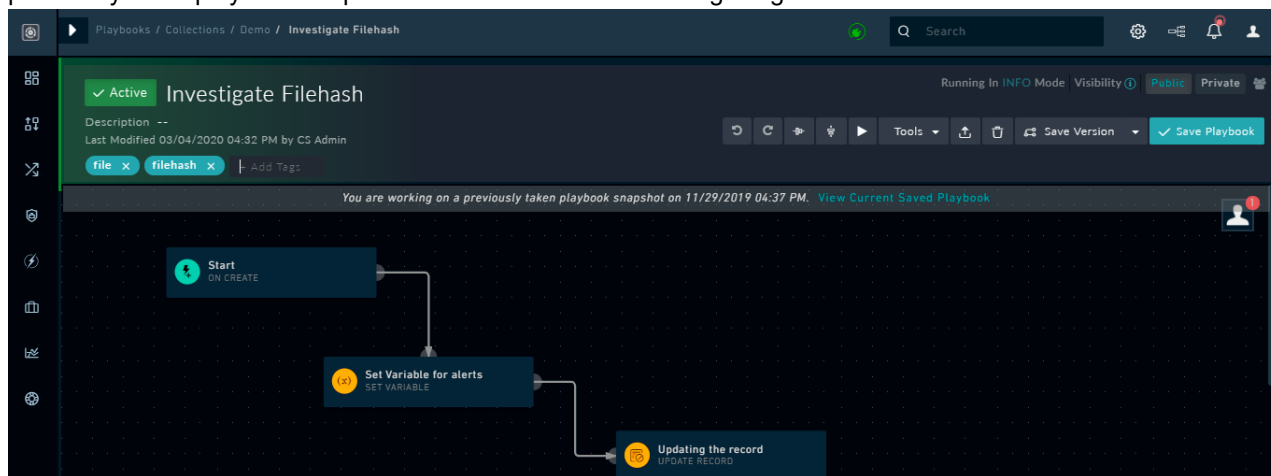
4. You can choose to view the playbook that is currently saved, by clicking the **View Current Saved Playbook** link, or you can **Save Playbook** to make this version the current saved version of the playbook and continue to work on the playbook.

Exporting versions of your playbook

You can choose to export playbook collections or playbooks with saved versions of the playbooks. This is extremely useful while developing playbooks, especially if you erroneously delete a step in the playbook or you want to go back to the previous state of the playbook. Retaining the versions of playbooks while exporting playbooks enables you to load a snapshot of a previously saved version of the playbook into an imported playbook

You can choose to export playbook collections or playbooks with saved versions of the playbooks as shown in the following image:
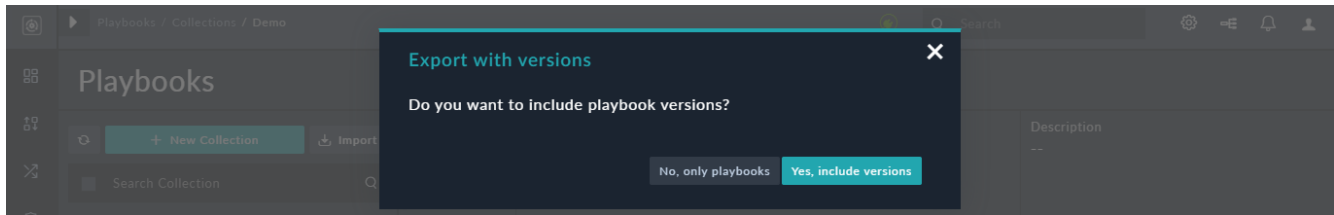
Clicking **Yes, include versions** on the above dialog will export playbooks or playbook collections with the saved versions of the playbook.

You can then import the playbook and then open that playbook in the playbook designer, you can see the previously saved versions of the playbook by clicking **Save Version** > **View Saved Versions**. This opens the `Versions` dialog as shown in the following image:
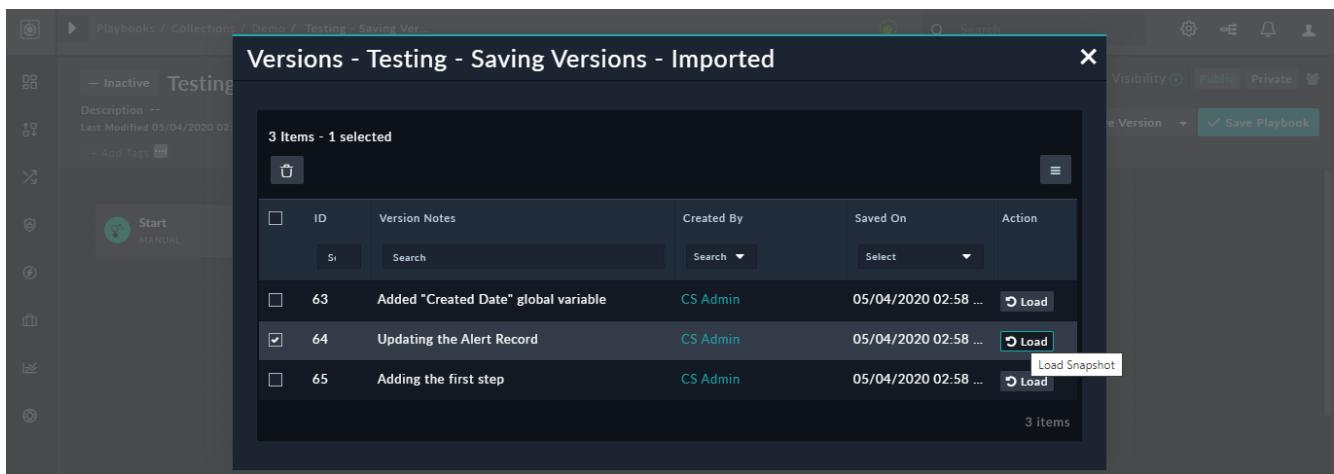
You can load a snapshot of a previously saved version of playbook in the `Versions` dialog by selecting the snapshot that you want to load in the playbook designer and clicking **Load**. This will display a message such as "You are working.....playbook snapshot...." as shown in the following image:
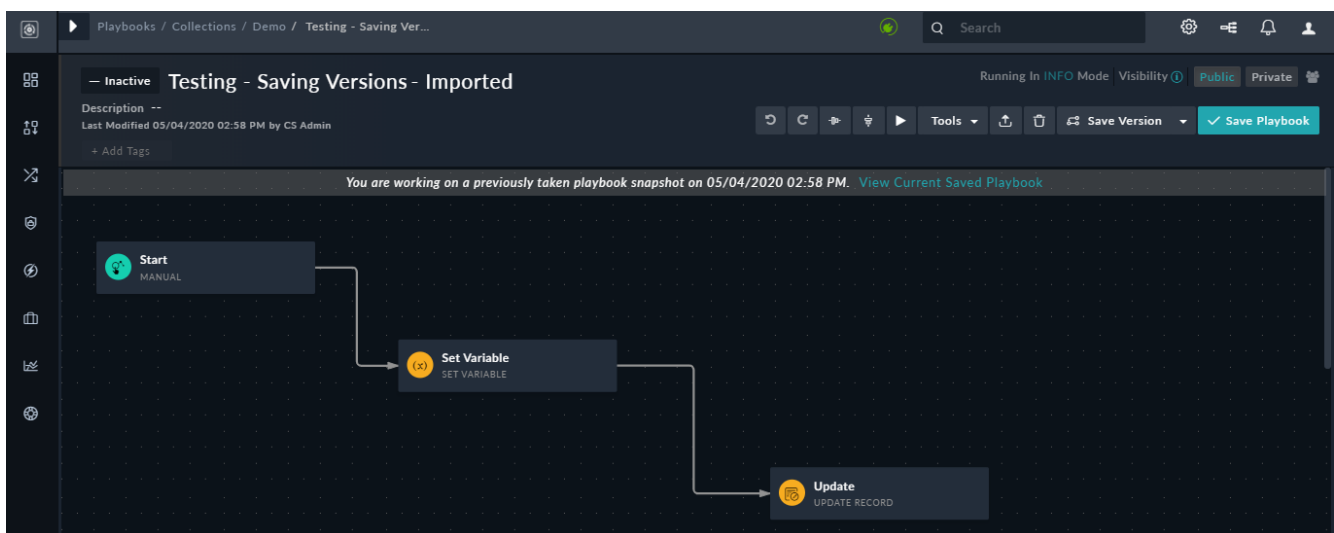
You can save this version of the playbook and continue to work on it or you can click **View Current Saved Playbook** to revert back to the state of the playbook when it was last saved.

Playbook recovery

FortiSOAR autosaves playbooks so that you can recover playbook drafts in cases where you accidentally close your browser or face any issues while working on a playbook. These unsaved (autosaved) drafts do not replace the current saved version of the playbook and only ensure that you do not lose any of your work done in the playbook, by providing you the ability to recover the drafts.

Playbook recovery in FortiSOAR is user-based, which ensures that users see their own unsaved drafts of the playbook. Since it is browser-based, it comes into effect as long as the same browser instance is used by the user. Also, playbook drafts might not be saved if you are working in the "Incognito" mode.

By default, FortiSOAR saves playbook drafts **15** seconds after the last change. However, you can ask your administrator to change this time across all playbooks by modifying the time, in seconds, on the **Application Configuration** tab in the `System Configuration` page. The minimum time that your administrator can set for saving playbook drafts is **5** seconds after the last change. You can also choose to disable (and later enable) playbooks recovery for all playbooks. For more information, see the *System Configuration* chapter in the "Administration Guide."

If the browser data is cleared, then the autosaved drafts will get deleted.

To recover an unsaved draft of the playbook, reopen that playbook in the playbook designer you will be prompted to confirm whether you want to recover the draft of the playbook as shown in the following image:

Once you click **Confirm** on the `Confirm` dialog, the autosaved version of the playbook is loaded in the playbook designer, and you can then choose to save this playbook using **Save Playbook** and make it the current working copy.

System Playbooks

FortiSOAR includes some system playbook collections that are used to automate tasks, such as the `Escalate` playbook which is used to escalate an alert to an incident based on specific inputs from the user and linking the alert(s) to the newly created incident.

You can also reference the `Escalate` system playbook from other playbooks.

> If you have referenced the `Escalate` system playbook in a referenced playbook, then in the records field, you must provide the input as {{vars.input.records}}, i.e., you need to pass the entire record and not the just the ID `{{vars.input.records[0]['@id']}}` as you would do for record fields in other cases.

Use the `Report Management Playbooks` collection system playbooks to that manage generating reports. For example, the `Generate Report By Scheduler` playbook generates reports based on schedules that you have specified.

The FortiSOAR UI includes links on the `System Configuration` page to the various playbook collections and templates, which are included by default when you install your FortiSOAR instance. Administrators can click the **Settings** (⚙) icon to open the `System Configuration` page and click the **System Fixtures** tab to access the system playbooks or fixtures. The `System Fixtures` page contains links to the system playbook collections and templates. Administrators can click these links to easily access all the system fixtures to understand their workings and make changes in them if required. To access System Notification and Escalation Playbooks, click the **System Notification and Escalation Playbooks** link.
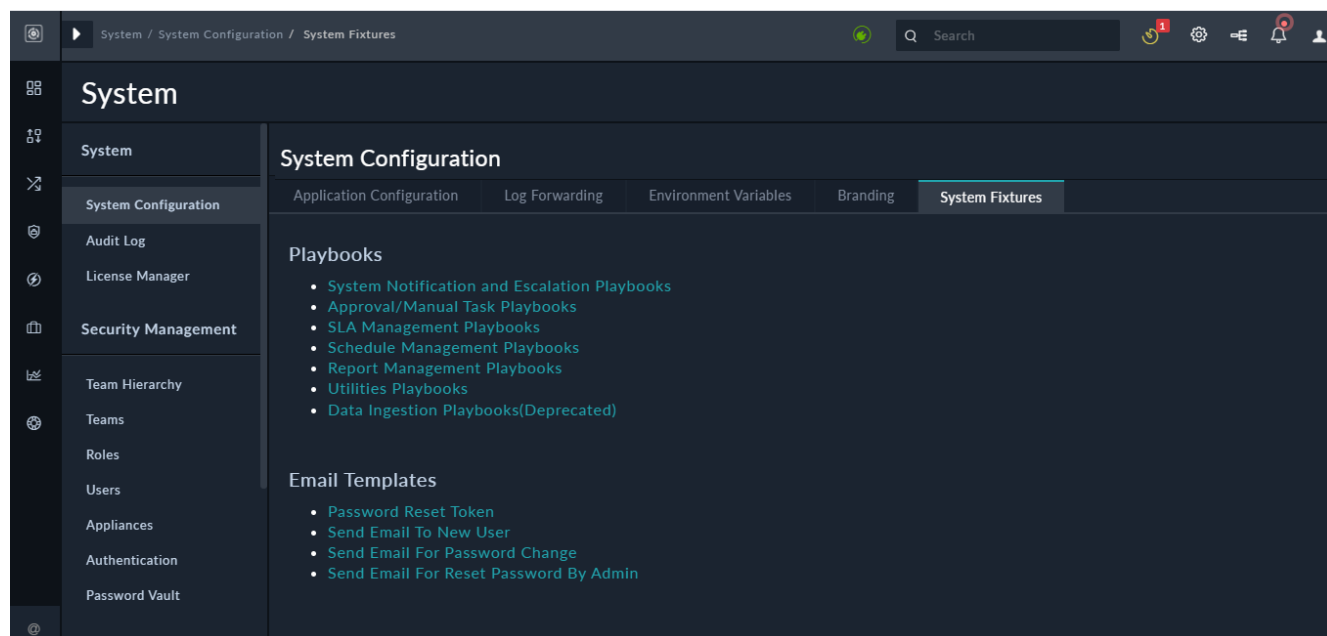
You can modify system playbooks as per your requirements.

> ⚠ Modifying any other fields in the notification playbooks or incorrectly modifying any system or SLA playbooks can affect FortiSOAR functionality.

For example, if you want to modify the default email signature, which is currently `Thanks, Fortinet`, for a system playbook for task notifications, you require to modify playbooks whose name contain `Notify`. For example, `Alert > Notify Creation (Email)`. To modify the email signature, open the playbook and double-click on the `Send Email` (Send User Email) and (Send Email To Admin Instead) step. In the `Send Email` step, in the **Content** field, modify the signature as per your requirements and click **Save**.

In the system playbook (or any playbook) that is sending an email, ensure that you have used the `Server_fqhn` global variable in the `Send Email` step.

When you are using a system playbook that sends an email, for example, when an alert is escalated to an incident, and an Incident Lead is assigned, then the system playbook (part of System Notification and Escalation Playbooks) sends an email to the Incident Lead specified. The email that is sent to the Incident Lead contains the link to the incident using the default hostname, which is the hostname that you had specified or that was present when you installed FortiSOAR. To ensure that the correct hostname is displayed in the email, you must update the appropriate hostname as per your FortiSOAR instance, in the playbook, using the Playbook Designer as follows:

1. Open the Playbook Designer.
2. Click **Tools** > **Global Variables** to display a list of global variables.
3. In the **Global Variables** pane, search for the `Server_fqhn` global variable, then click the `Edit` icon in the `Server_fqhn` global variable, and in the `Field Value` field add the appropriate hostname value.
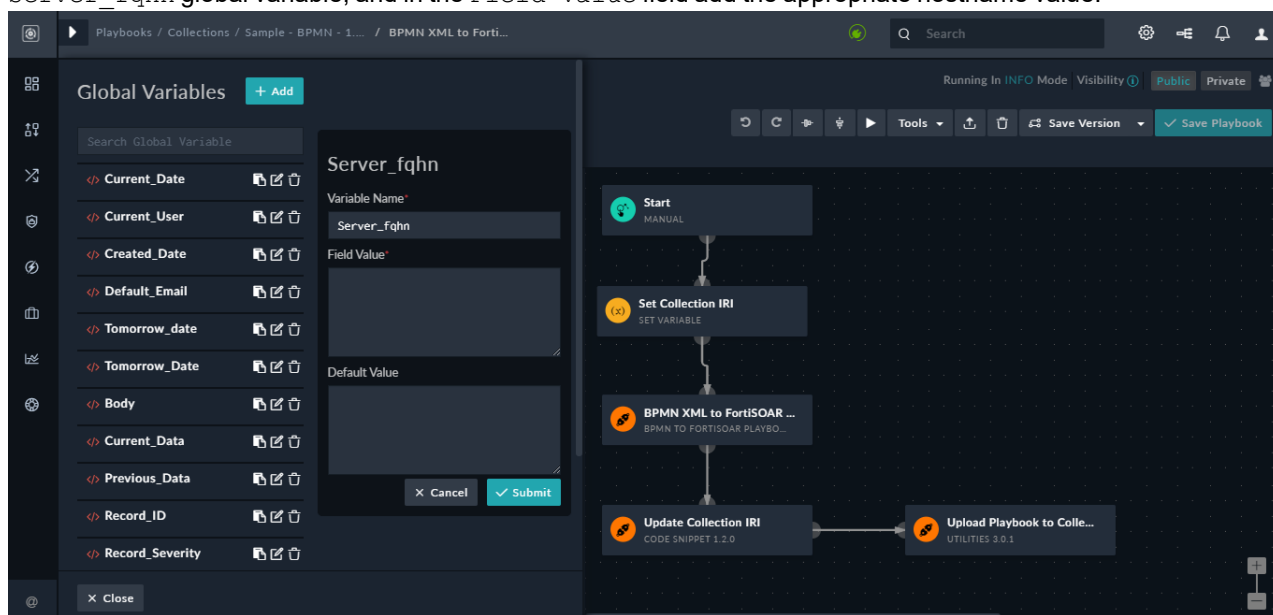
You can optionally specify a default hostname value in the `Default Value` field.

4. Click **Submit**.
   This adds the updated hostname for your incident and then when a system playbook sends an email the link contains the correct hostname.

SLA Management Playbooks

Use the SLA Management Playbooks collection to manage your SLAs. To access SLA Management Playbooks, on the System Configuration page, click the **System Fixtures** tab and then the **SLA Management Playbook Collection** link

SLA Management Playbooks auto-populates date fields in the following cases: when the status of incident or alert records have been changed to **Resolved** or **Closed** or when incident or alert records are assigned to a user. Details are as follows:

For **Alert** Records:

- When the status of an alert record is changed to **Closed**, **Closed (False Positive)**, or **Closed (Duplicate)** then FortiSOAR auto-populates the **Resolved Date** field in the alert record, with either today's date or the date on which the status of the record was changed to Closed, Closed (False Positive), or Closed (Duplicate).

- When an alert record is assigned to a user, then FortiSOAR auto-populates the **Assigned Date** field in the alert record, with either today's date or the date on which the record was assigned.

For **Incident** Records:

- When the status of an incident record is changed to **Resolved**, then FortiSOAR auto-populates the **Resolved Date** field in the incident record, with either today's date or the date on which the status of the record was changed to Resolved.
- When an incident record is assigned to a user (incident lead), then FortiSOAR auto-populates the **Assigned Date** field in the incident record, with either today's date or the date on which the record was assigned.

For information about all the system playbook collections and templates, which are included by default when you install your FortiSOAR instance, see the *System Configuration* topic in the "Administration Guide."

Triggers & Steps

Triggers

Triggers define when a Playbook is to be executed. Triggers are always the first step in a playbook. Once a playbook has been triggered, it flows through the remaining defined steps as defined by the routes on the canvas using the trigger as the starting point.

Trigger Types

There are six different trigger types defined in the Playbook Engine. Most triggers are based upon actions that you can perform on models in the FortiSOAR database. The parameters of each are defined below.

Once you add a playbook, the playbook gets created with a placeholder **Trigger** step as shown in the following image. Then specify the required parameters for the trigger and then click **Save** to add the first step to the playbook. The procedure for creating playbooks is mentioned in the *Playbooks Overview* section.
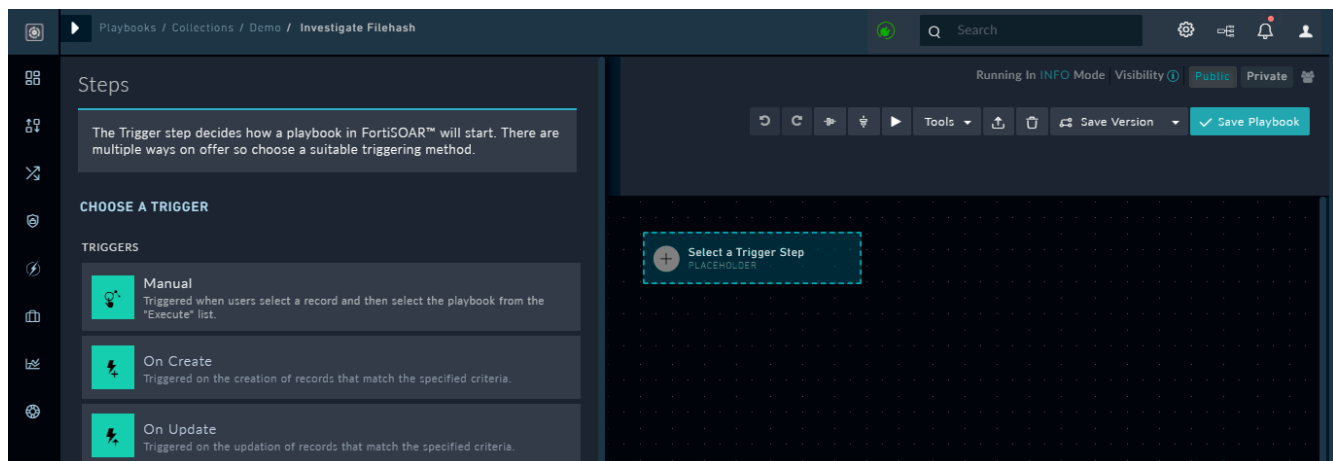
You can add **Step Utilities**, i.e., Variables and Messages for all triggers. Add variables for all trigger by clicking the **Variables** link that appears in the playbook step footer to add input variables. Input variables are the inputs that are required to be provided by the user at the time of playbook execution. Required variables are made available in the environment based on the given name. Required variables can be of any standard field format within the UI, including text, picklist values, lookup, and checkboxes. See Variables for more information. You can also add a custom message for each playbook step to describe its behavior. See Message for more information.

On Create Triggers

On Create triggers are intended for asynchronous execution, meaning they are non-blocking on the triggering data operation. For example, you can define a playbook that gets triggered when an Incident is created.

This trigger starts the execution of a playbook immediately after a record of the selected model type is created or ingested. Click **On Create Trigger** in the `Playbook Designer`, type the name of the step in the **Step Name** field and then select the module on whose creation you want to trigger the playbook, from the **Resource** drop-down list, for example, `Incidents`, and click **Save**.

You can also add conditions based on which you can trigger this playbook. For more information, see Condition-based triggers.

Nested filters are also supported on the "On Create" and "On Update" triggers. Support has also been added for Less Than (Before in case of Date/Time fields), Lesser Than or Equal To (On or Before in case of Date/Time fields), Greater Than (After in case of Date/Time fields), Greater Than or Equal To (On or after in case of Date/Time fields), and Matches Pattern operators in filters. For more information about nested filters and operators, see the *Dashboards, Templates, and Widgets* chapter in the "User Guide."

On Update Triggers

This trigger starts the execution of a playbook immediately after a record of the selected model type is updated. You can create an On Update trigger on almost all models, and can add an On Update trigger in the same way you add an On Create trigger. An update could be made to any field within the model, **including linking or changing one or more new relationships**.

When you add the **On Update** trigger to run on a **Is Changed** condition when relation fields are changed, such as indicators for alerts, then the **On Update** trigger will trigger the playbook only when the related record is linked from the same side. For example, while linking an indicator to an alert, the relation can be formed both ways – by updating the indicator record and linking the alert; or by updating the alert record and linking the indicator.

However, an On Update trigger on an alert when indicator 'Is Changed' will only be triggered if the *indicator was linked by updating the alert record*. It will not be triggered when the relation is established while creating or updating an indicator record.

The single update action defines the trigger, so linking multiple records or updating multiple fields simultaneously does not trigger the playbook multiple times. However, multiple inline edits trigger the playbook multiple times. A bulk edit action triggers the Playbook only once.

You can also add conditions based on which you can trigger this playbook. For more information, see Condition-based triggers.

On Delete

This trigger starts the execution of a playbook immediately after a record of the selected model type is deleted. You can create an On Delete trigger on almost all models, and can add an On Delete trigger in the same way you add an On Create trigger.
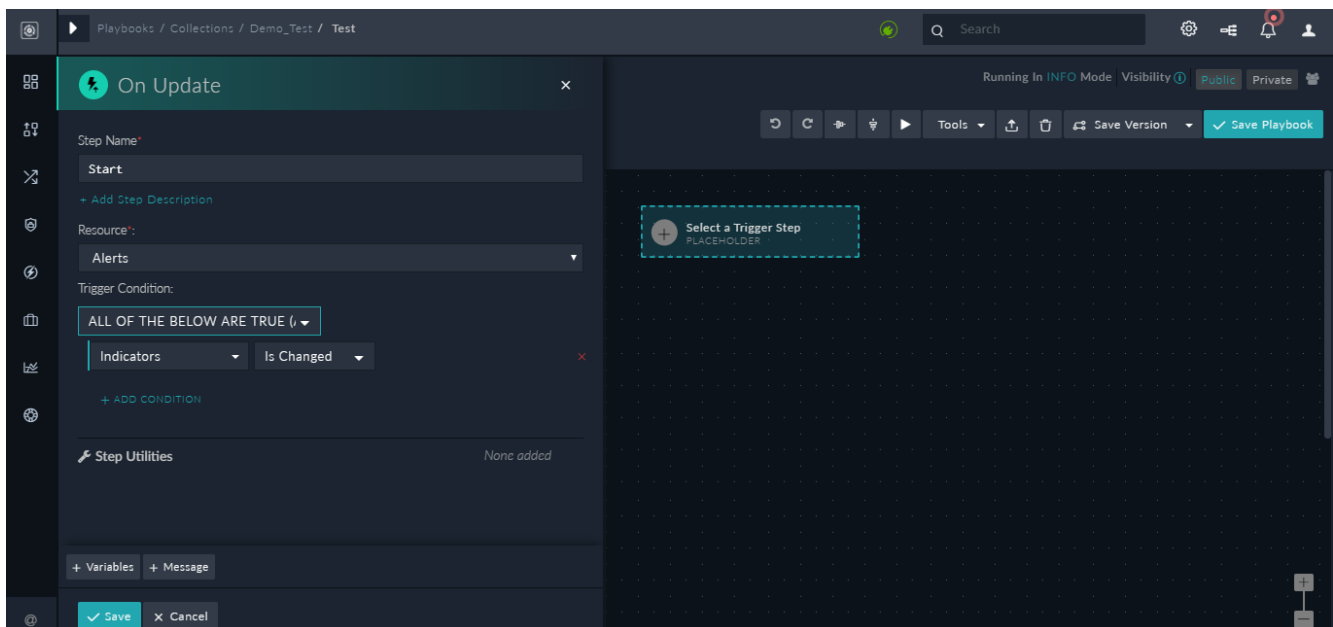
You can also add conditions based on which you can trigger this playbook. For more information, see Condition-based triggers.

Condition-based triggers

You can define a condition or nested conditions to trigger a playbook only if the specified filter criteria are met. This streamlines playbook calls and prevents the excessive calling of playbooks.

> You cannot apply filters on encrypted fields.

Open the playbook designer and click on On Create, On Update, or On Delete trigger. For example, click **On Update** trigger and then select the module, which when updated will trigger the playbook, from the **Resource** drop-down list, for example, `Alerts`. Once you select the resource, a **Trigger Condition** drop-down list appears. To define the condition based on which the decision to trigger the playbook will be taken, perform the following steps:

1. From the **Trigger Condition** drop-down list, select the logical condition, **All of the below are True (AND)**, or **Any of the below is True (OR)** to trigger the playbook.
   In case of the **AND** condition the playbook gets triggered only if all the conditions specified are met. In case of the **OR** condition the playbook gets triggered if any of the conditions specified are met. The **AND** or **OR** conditions are mutually exclusive, i.e., you can only choose one of them to apply to conditions.
2. Click the **Add Condition** link and then build your condition.
   Note: There is an additional operator **Is Changed** added for the trigger condition. If you select the **Is Changed** operator for a field, then the playbook will be triggered whenever the specified field is changed.
   For example, if you want to assign **Critical** alerts that are in the **Open** state to a specific user, say `csadmin`, then you can select the **Severity** field and choose the operator as **Equals** and specify **Critical**.
   Click the **Add Condition** again to define another condition and then select the **Status** field and choose the operator as **Equals** and specify **Open**, as shown in the following image:
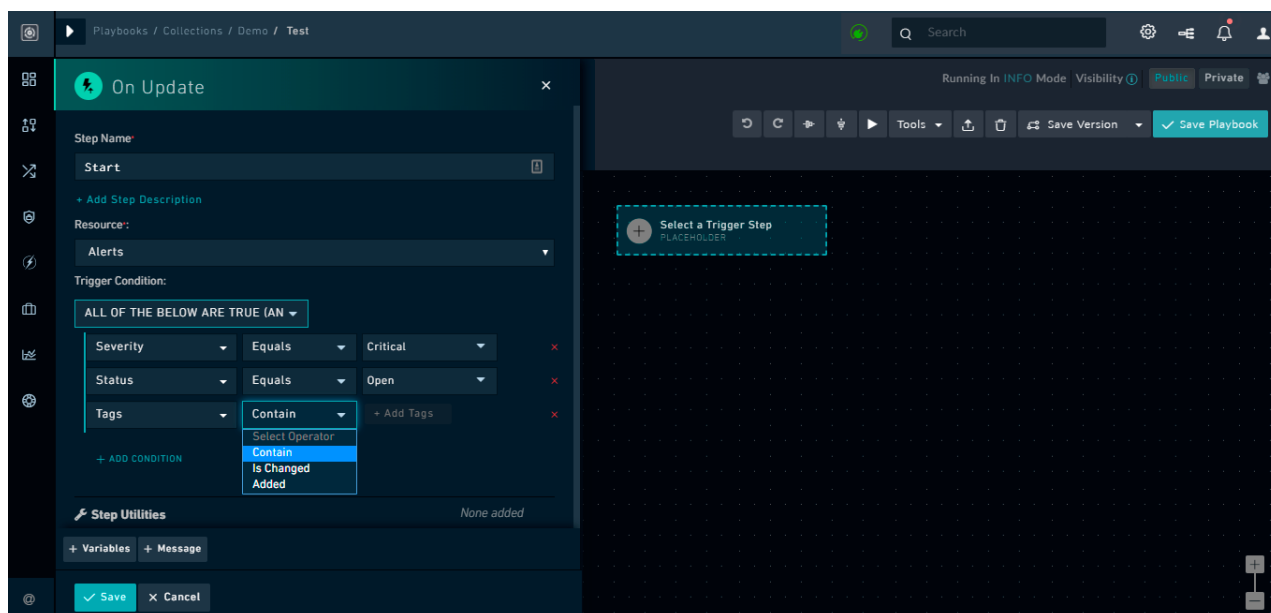
Once you complete adding the conditions, click **Save** to save the playbook.

In this case, once the condition is met, the On Update playbook will be triggered, and based on the steps that you have defined, for example, the `Update Record` step, the alert will be updated and assigned to `csadmin`.

Important: You can also use `Tags` as a condition to trigger the On Create, On Update, or On Delete playbooks. You can add special characters and spaces in tags; however, the following special characters are not supported in tags: `'`, `,`, `"`, `#`, `?`, and `/`. The operators that you can use with **Tags** in the **On Create** and **On Delete** triggers are `Contains`. The operators that you can use with the **On Update** trigger are `Contains`, `Added`, or `Is Changed`.

If you want to add a group of conditions, then click the **Add Conditions Group** link. For example, if you wanted to create a condition where the alerts have been created in the last calendar month and whose severity is critical and whose status is open or investigating, in such a case you could create a condition group for the status condition. For more information about nested filters and operators that can be used in conditions, see the *Dashboards, Templates, and Widgets* chapter in the "User Guide."

Custom API Endpoint

Custom API Endpoint Triggers allow you to specify an arbitrary endpoint that can be used to externally start a playbook using a REST API POST action from another system. All playbooks are triggered using an API on a technical level with the microservices architecture used by the application, but conceptually, this trigger allows for the creation of an endpoint explicitly for use in API-based operations.

The chief aim of the Custom API Endpoint Trigger option is to allow for easy ingestion of data. A RESTful POST method explicitly defined by the authentication method is allowed to trigger a playbook to the defined endpoint. The endpoints of the Custom API Endpoint trigger are not discoverable, unlike the standard API routes within the JSON-LD / Hydra definition. You must know the endpoint name explicitly, and it currently only allows the POST method.

The endpoint name can be any valid name using alphanumeric characters. You should not use special characters in naming the endpoint, or the endpoint might not function correctly.

The following three types of authentication are currently supported:

1. Token-Based (default) - the default API method for signing any API request.
   Note: For token-based (HMAC) authentication the timestamp must be in UTC format.

2. Basic Authentication - a Base64 encoded version of the `username:password` present in the header. This requires the username and password of a user without 2-Factor Authentication turned on to properly function. Note that this method also uses a separate endpoint.

3. No Authentication (Not recommended) - no authentication method is applied to the endpoint, and any RESTful POST method will trigger the playbook. This is chiefly aimed at applications where the only option for exporting data is by using a webhook, but this method is not recommended for routine usage due to the lack of security.

You can manually create your own security method with this trigger by defining a specific criterion to be used in a Decision Step verifying information in the full Request blob.

To add an Custom API Endpoint trigger, click **Custom API Endpoint** trigger in the `Playbook Designer`, type the name of the step and the API route in the **Step Name** and **Route** fields respectively, and then select the `Authentication Method` from the ones specified earlier and click **Save**.

Referenced

The Referenced trigger is intended for playbooks that are exclusively called from a `Reference a Workflow` step, which is discussed in a later section. Bear in mind that any dynamic data requirements must be made available from the Parent (s) playbooks to be used during the execution of a Child playbook.

To add Referenced step, click **Referenced** in the `Playbook Designer`, type the name of the step in the **Step Name** field and click **Save**.

Manual Trigger

The Manual Trigger allows you to call a specific playbook from within any module in the system, i.e., these are are for click-to-start playbooks. You can then execute any desired operations within that playbook on demand.

To add a Manual trigger, click **Manual Trigger** in the `Playbook Designer`, type the name of the step in the **Step Name** field. In the **Trigger Label Button** field, type the name that will be displayed in the selected module (s) to trigger this playbook. The name that you specified in this field is what the user will see in the **Execute** drop-down list on the module list.

The Manual Trigger step provides you with options to specify whether the execution behavior of the playbook, i.e., you can decide whether the playbook requires a record to be executed or if it does not require a record to be executed. If the playbook requires a record to be executed, then select the **Requires record input to run** option and then select the run mode, i.e., if the action must be executed once, then select the **Run once for all selected records** option or if the action must be executed separately for each selected record then select the **Run separately for each selected Record**. By default, the **Run once for all selected records** option is selected. This makes it more effective to handle multiple selections since you do not require to write two playbooks and map the second playbook in the first playbook.

> In the case of a "Manual Trigger" step that has the **Run separately for each selected records** option selected, and in which you have selected multiple records and triggered a playbook from the designer for debugging purposes, you will observe that only a single playbook will be triggered on a single record to simulate the output. For information on triggering playbooks from the playbook designer for debugging, see the *Playbook Debugging - Triggering and testing playbooks from the Designer* topic in the Introduction to Playbooks chapter.

If you want the playbook to run without having to select a record, then select the **Does not require input record to run** option. This acts as a module-based trigger, i.e., you can trigger a playbook based on a selected module without having to select a record in the specified module. An example of this could be a manual trigger to check for new alerts from a SIEM tool could be run globally on the `Alerts` module.

In either of the cases, from the **Choose record modules on which the playbook would be available on** select one or more modules on which you want to register this trigger and execute the playbook. For example, you can choose `Alerts` and `Incidents`. When you select this Manual Trigger from the **Execute** drop-down list, the playbook gets executed, and at the time of execution, the record (s) of the registered module (s) are passed into the playbook environment with the trigger.
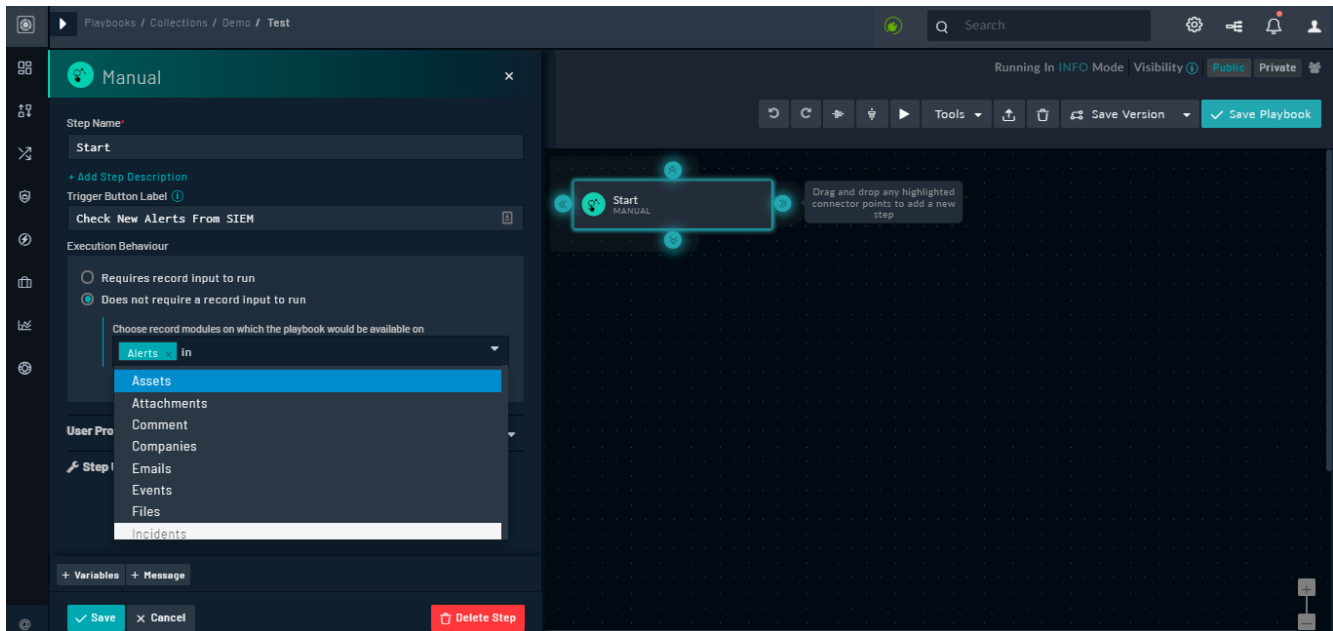
The playbook that you create with the **Does not require input record to run** option will appear in the **Execute** drop-down list in the module, or you can also create a specific button for this action, by updating the module template. In case of our example, when you open the **Alerts** module, you will see the **Check New Alerts From SIEM** option in the **Execute** drop-down list in the module (when you do not select any record), as shown in the following image:
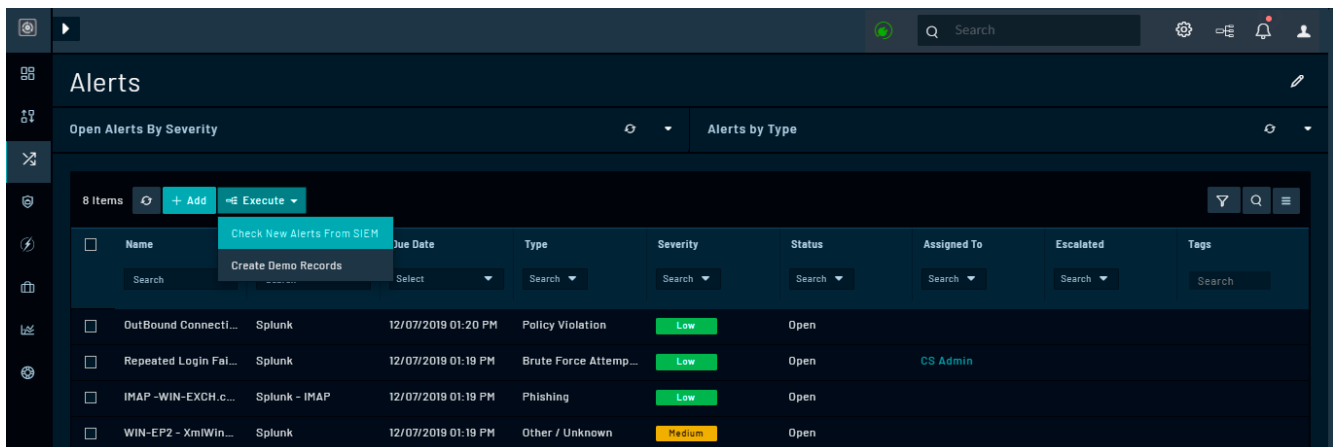
However, if you select a record in the Alerts module, then you will observe that the **Check New Alerts From SIEM** option will not present in the **Execute** drop-down list.

When you choose **Run once for all selected records** in the `Execution Behaviour` section, then a single playbook is run with the input set as `vars.input.params.records`, which is an array containing a list of all selected records that acts as an input to the playbook. When you choose **Run separately for each selected record** in the `Execution Behaviour` section, then one instance of the playbook is run per selected record with the input to the playbook set as `vars.input.params.records`.

Once you have completed providing all the above parameters, click **Save** to save the Manual Trigger.

When you want to execute a playbook action, click the module (s) on which you have registered the Manual Trigger. This would be a module you have specified in the **Choose record modules on which the playbook would be available on** field. In the grid view of this module, click one or more records to display the **Execute** drop-down list (if you have selected the **Requires record input to run** option). Pressing the down arrow provides a list of available actions. The name of the action displayed is based on the name that you have specified in the **Trigger Label Button** field, for example, if you have entered `Investigate Phishing eMail` in the **Trigger Label Button** field, then **Investigate Phishing eMail** is an option in the **Execute** drop-down list as shown in the following image:
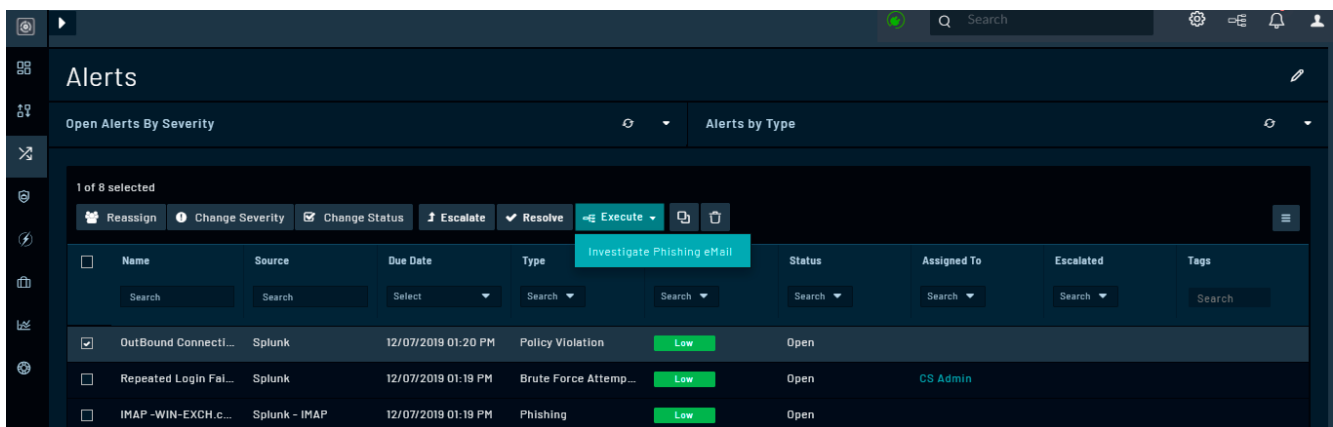
An example of a Manual Trigger that is included by default in the `Alerts` module is the **Escalate** action. Select a record in the Alerts module and click **Escalate** to automatically create a new incident based on the inputs you provide in the `Escalate` dialog and also link the alert(s) to this newly created incident.
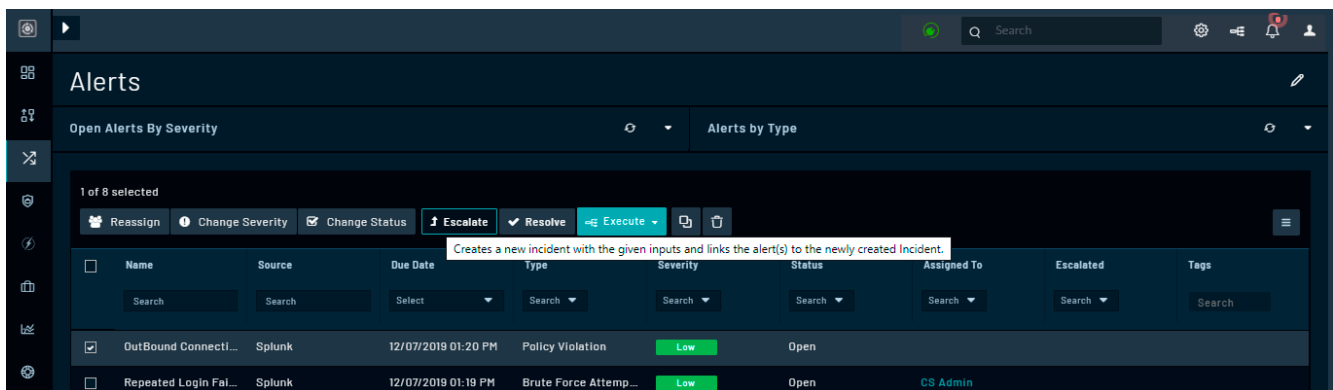
When you initiate an action with an associated required input variable, such as the **Escalate** action, you will be prompted to enter that information in an input dialog as shown in the following image:
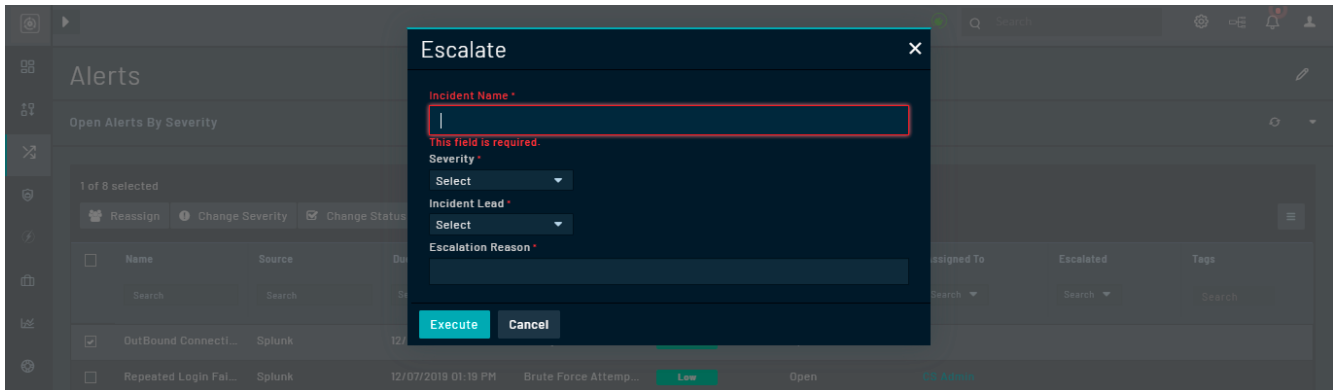
Enter the required information and click **Execute** to execute the Escalate playbook.

You can also define visibility conditions for those playbooks that require record input to run. You can define conditions on records, such as the specific record type or severity or status; thereby enabling users to see only those actions (playbooks) that apply to records that match the defined condition. For example, a `Submit Malware Sample` playbook should be visible for a "Malicious Code" incident, but it should not be visible for an "Unauthorized Access" incident.

In the `Execution Behaviour` section, if you have selected the **Requires record input to run** option, click the **Configure Visibility Conditions** link to add the visibility conditions as shown in the following image:
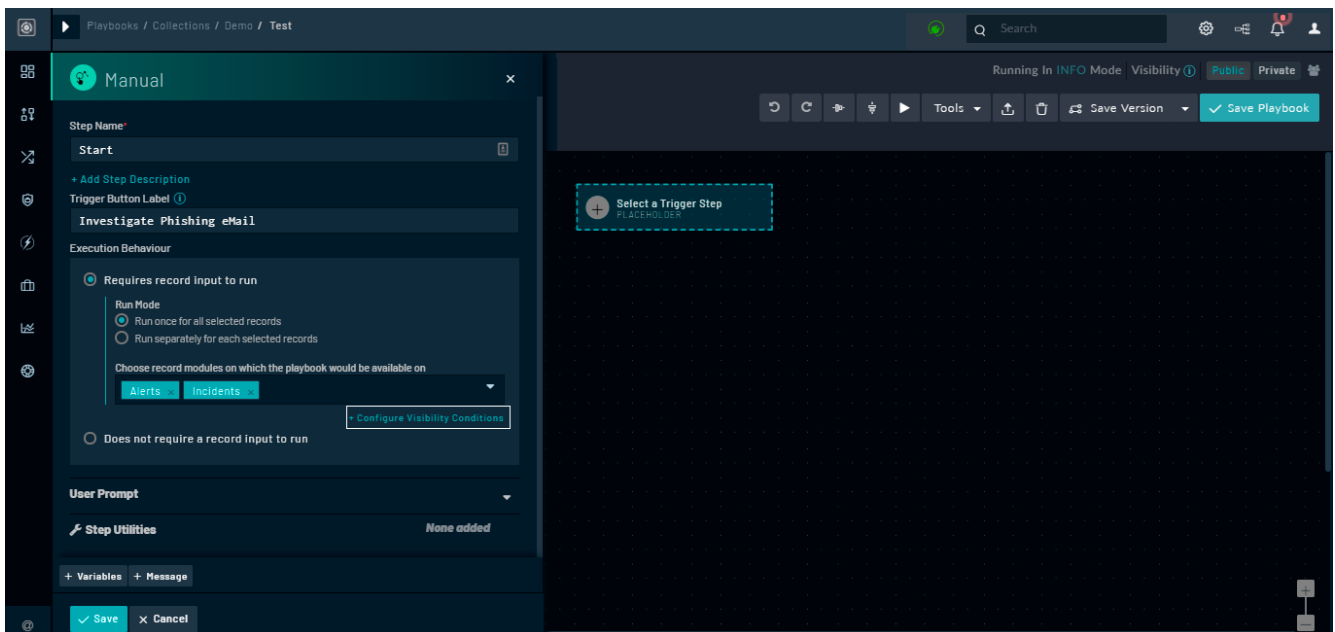
You can define distinct conditions for each selected module in the playbook, as separate sections for each selected module is displayed; thereby allowing you to apply different display conditions (filters) for each selected module as shown in the following image:
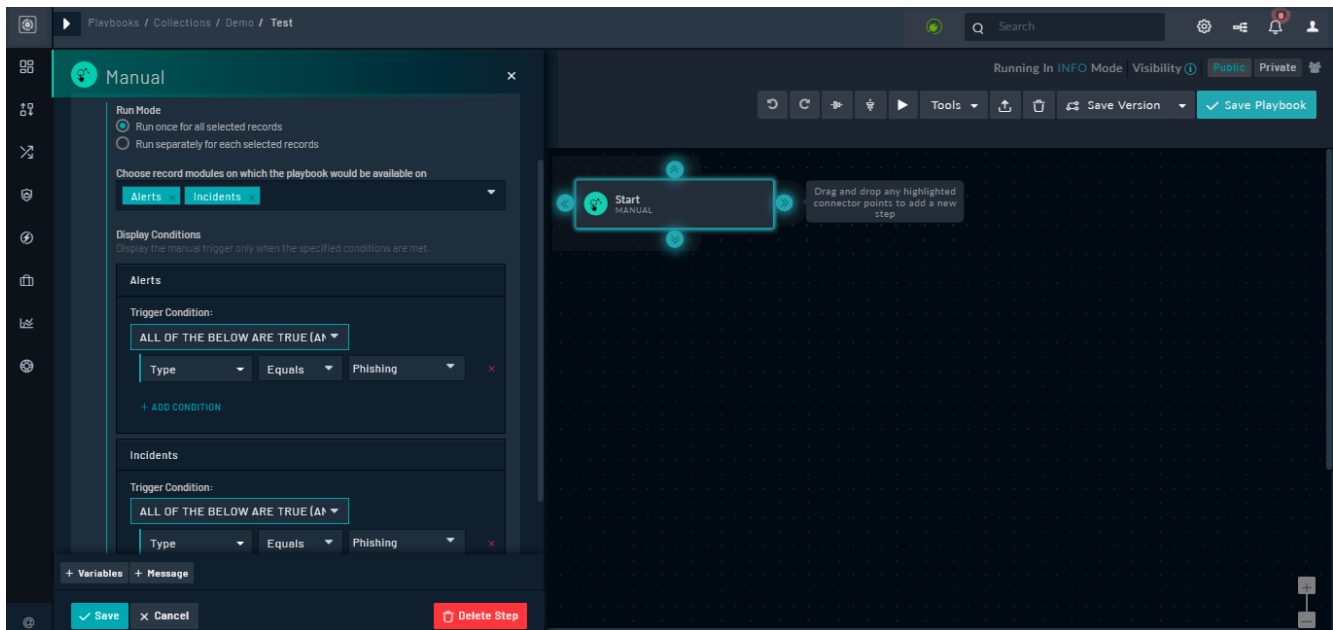
In the above image **Alerts** and **Incidents** modules are selected in the **Choose record modules on which the playbook would be available on** list, and therefore the `Display Conditions` section contains trigger conditions for both the Alerts and Incidents modules. If you do not specify any display conditions, then all playbooks that you have defined for the modules can be viewed when you select or open a record in the **Execute** list. An example of a condition based on which a playbook is displayed when a user selects a record would be a playbook that is defined to be run only on "alerts or incidents of type phishing." In this case, in the manual trigger step, in the **Trigger Button Label** field you could type `Investigate Phishing eMail`, and in the `Display Conditions` section, you would define a **Trigger Condition**`Type Equals Phishing` for both the Alerts and the Incidents modules as shown in the above image.

> If you add a filter in a `Trigger Condition`, with an `Equals` or `Not Equals` logical operator to a richtext content field, such as **Description**, then you must enclose the content you want to filter in `<p>...</p>` tags.

Once you define this condition, then users will see this playbook only when the alert record is of type phishing. For example, you have alert records: `Alert 7` whose type is set to Phishing, and `Alert 8` whose type is set to Policy Violation. When you click the **Alert 7** alert record to view its details, in the **Execute** list you can see the `Investigate Phishing eMail` playbook listed, as showing in the following image:
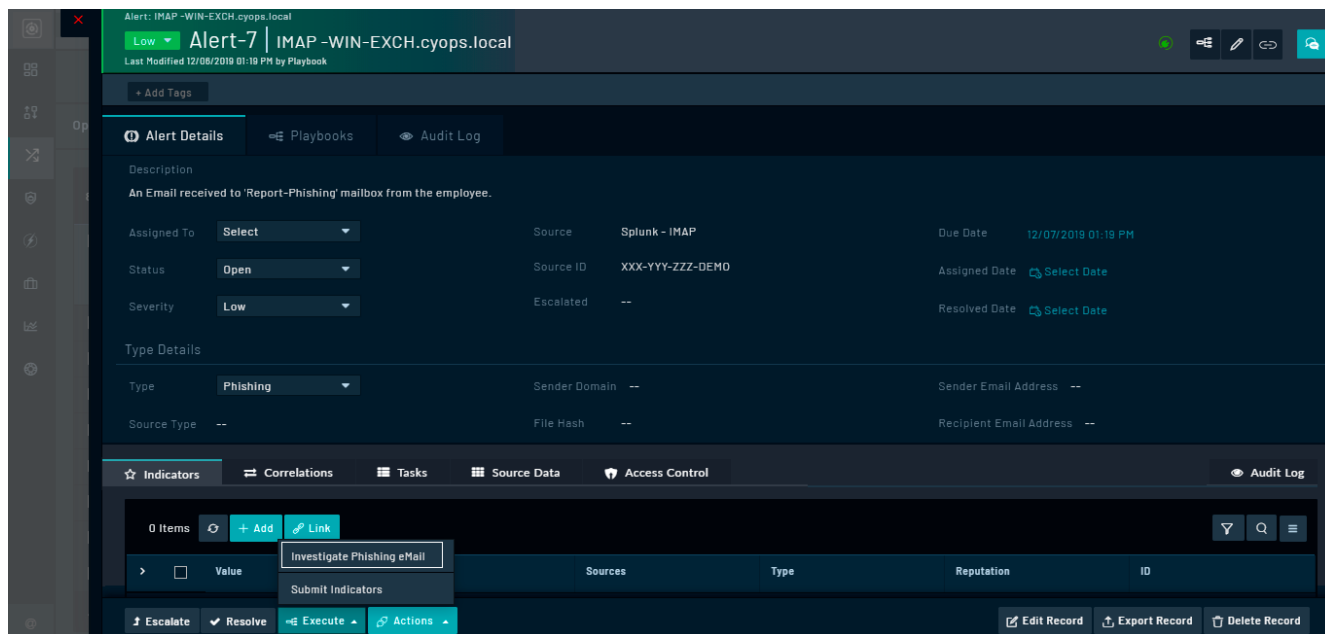
However, when you click the **Alert 8** alert record to view its details, in the **Execute** list you will not be able to see the
`Investigate Phishing eMail` playbook as seen in the following image:
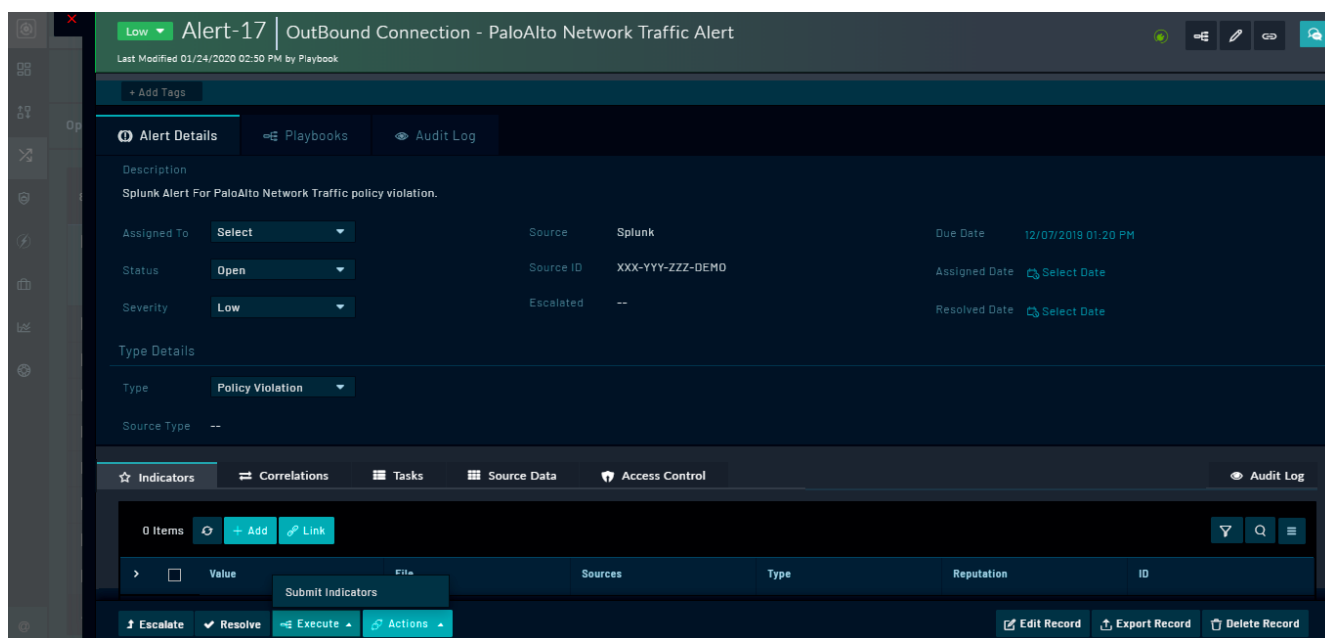
Building a User Prompt

You can build a customized user prompt form by adding multiple types of input fields of standard field format within the UI
such as Text, Picklist, Lookup, File, Phone, Integer, Decimal, Date/Time, Dynamic list, Checkbox, and Email. If you
select the field format as **Text**, you can also define its **Sub-type** such as Text Field, Domain, Rich Text, etc. Click **Add
Field** in the `User Prompt` section, to add an input field to build your user prompt. The User Prompt enables you to
create a customized user prompt.

You can now configure the following in the User Prompt:

- Specify field titles and variable names, instead of having the field title being built automatically.
- Add tooltips for fields.
- Change the action button name to a name of your choice; the default is `Execute`.
- Display a pre-populated form field in the input form for review or modification, before executing any action. One benefit of this feature is the ability to review certain fields that will be used in the playbook, such as a source IP address or closure notes.

You can build the user prompt using custom fields and fields from input record, if you have selected a single module, (e.g., Alerts) and not when you have selected multiple modules. If you select multiple modules, then you can build the user prompt using only custom fields.

The `User Prompt` section also contains an additional field where you can specify default values. The values entered in this field would be displayed when the `User Prompt` is shown to a user. You can either specify any custom value (if your input type is selected as **Custom**) or any default record field (if your input type is selected as **Record Field**). The listing of record fields will be based on the module that you have selected in the **Choose record modules on which the playbook would be available on** field. Once you select the record field, then the data of this field will be loaded from the specified input record and displayed to the user in the `User Prompt`.

The default values will not update the record; they are only used to display content in the User Prompt

---

An example of building a user prompt in FortiSOAR will be if you want to reassign a number of alert records to another user after specifying a note. This example will also demonstrate how you can use custom field titles and variables and customizing the **Execute** button name. This example assumes that you have selected **Alerts** from the **Choose record modules on which the playbook would be available on** field.

Steps to create this example user prompt is as follows:

1. In the `User Prompt` section, click the down arrow, and then click **Add Field**.
2. To reassign the alert record to another user by using an input record (Assigned To) do the following:
   a. From the **Input Type** drop-down list, select **Record Field**.
      Note: If you have selected multiple modules in the **Choose record modules on which the playbook would be available on** field, you cannot select Record Field from the Field Type drop-down list and you can create the User Prompt using custom fields only.
   b. From the **Choose Record Field** drop-down list, select the field that will be set by default.
      The field listing in the **Choose Record Field** drop-down list is dependent on the module you have selected in the **Choose record modules on which the playbook would be available on** field. For our example, we have chosen **Alerts**.
      For our example, select **Assigned To**.
   c. You can choose to select whether this field will be mandatory or not in the user prompt, by selecting or clearing the **Mark as Required Field In Prompt** checkbox.
   d. In the **Field Label** field, type the label of the field that will be displayed in the User Prompt.
      For example, `User Assignment`.
      The **Variable Name** field type gets auto-populated with the variable name, for example, `userAssignment`. You can edit the variable name if you want.
      Important: If you are using Dynamic Values in the next step of the playbook note that Dynamic Values will display *custom parameters* in the **Input > Parameters** option, and *Input Record Fields*, such as `AssignedTo` in the **Input > Records** option.

e. (Optional) If you want to provide more information about the field, then click the **Add Tooltip** link and enter the description in the **Tooltip** field.

3. To create a custom field for providing a reason or notes for the reassignment, do the following:

   a. Click the **Add Field** link.

   b. From the **Input Type** drop-down list, select **Custom**.

   c. From the **Field Type** drop-down list, select **Text**.
   Note: If you select the field type as "Text", you can also choose its **Sub-type**, such as Rich Text, Text Area, IP, etc. Also, if you select the field type as "Picklist" then you must select the corresponding picklist, and if you select "Lookup", then you can specify the related module.

   d. You can choose to select whether this field will be mandatory or not in the user prompt, by selecting or clearing the **Mark as Required Field In Prompt** checkbox.
   For our example, we will click the **Mark as Required Field In Prompt** checkbox, to ensure that the record cannot be reassigned to another user without adding a note.

   e. In **Field Label** field, type the label of the field that will be displayed in the User Prompt.
   For example, `Notes for Reassignment`.
   The **Variable Name** field type gets auto populated with the variable name, for example, `notesForReassignment`. You can edit the variable name if you want.

   f. (Optional) In the **Default Value** field, you can enter the default value for the custom field.
   Note: You can specify either a "Static" date/time or a "Custom" date/time as a default value, if your custom field is of type "Date/Time". If you select **Static**, click the **Select Date** icon to display the Calendar and select the

required date/time. If you select **Custom**, then you can specify a date/time relative to the current date/time such as 1 hour from now, or 3 hours ago.

    g. (Optional) If you want to provide more information about the field, then click the **Add Tooltip** link and enter the description in the **Tooltip** field.

4. To change the name of the action button, which by default appears as Execute, update the **Submit Button Text** field, and type, for example, `Reassign.`

The `Input Prompt` section with all these changes will appear as shown in the following image:

Now, when you execute this playbook after selecting alert records and clicking **Execute** > **Reassign with Notes**, then the `Reassign with Notes` dialog is displayed. The `Reassign with Notes` dialog will contain the **Assigned To**

drop-down list, with a list of users to whom this record can be reassigned, a rich text area where the user must add the reassignment notes, and the **Reassign** button which will execute this playbook, as shown in the following image:
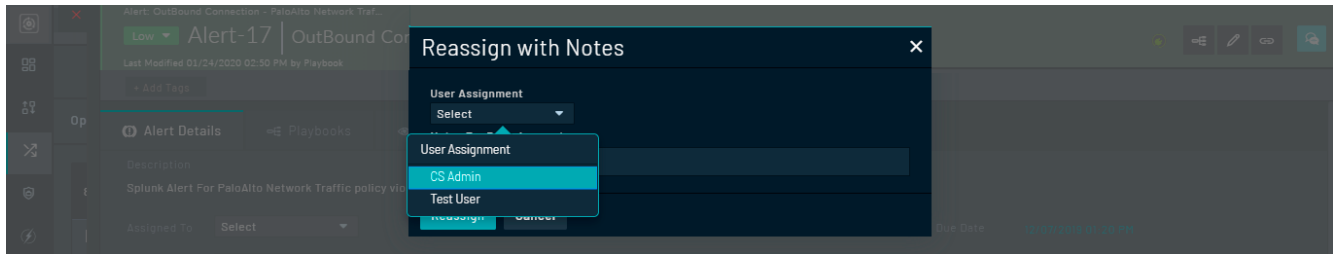
User Input Prompt - Visibility Conditions

From version 7.0.0 onwards, you can add visibility conditions to the fields displayed in the user input form, i.e., fields in the user form would be visible based on the conditions you specify. You can define visibility conditions in user prompts both when you trigger the playbook using the *Manual Trigger* option and also during the execution of the playbook using the *Manual Input* step (Input-based user prompt).

For example, when you trigger a playbook on an alert record, you could ask users to specify the type of alert, and you could define additional fields that would be visible if a particular type of alert is selected. For example, if the user selects the 'Phishing' as the alert type, then another field named 'Phishing Type' would be displayed, if the user selects 'Ransomware' as the alert type, then a field named 'Ransomware Type' would be displayed and so on. Based on the user selection, you can further define the playbook execution.

To add visibility conditions as the one described in the example, i.e., display an additional field based on the type of alert, in a User Prompt, do the following:

1. In the `User Prompt` section, click the down arrow, and then click **Add Field**.
2. To prompt the user to set the 'Type' for the alert, select the **Input Type** as **Record Field** . From the **Choose Record Field** drop-down list select **Type**, click the **Mark as Required Field In Prompt** checkbox, and in the **Field Label** field, enter `Type`.
3. Click **Add Field** to create additional fields based on the Type of alert the user selects. For example, to create a 'Phishing Type' field, from the **Input Type** drop-down list, select **Custom**, from the **Field Type** drop-down list, select **Text**, and from the **Sub Type** select **Text Field**. If you have created a picklist with the different Phishing Types, you can specify Picklist and choose the appropriate picklist. Next, click the **Mark as Required Field In Prompt** checkbox, and in the **Field Label** field, enter `Phishing Type`.
   Note: If you add a field as required, for which a visibility condition is defined, then that field is required only when its visibility condition is met, i.e., when the field is visible. For example, in the above step, the Phishing Type field is a required field, however, this field will be required only if the Type of alert is 'Phishing'.
   You can similarly add fields for various types of alerts, such as Ransomware, or Brute Force Attack, etc.
4. Add the visibility condition for the `Phishing Type` field by clicking **Add Visibility Conditions** and specifying the **Visibility Condition** as "`Type Equals Phishing`":

Similarly, you can other visibility conditions for various steps. To define a visibility condition there must be at least two steps in the user prompt.

5. Click **Save** to save your changes to the step, and then click **Save Playbook**.

When you run the playbook on a record, you will observe that if you select 'Phishing' as the type, the 'Phishing Type' field is displayed. If you select any other type, you will observe that no additional field is displayed.

User Input Prompt - Dynamic Lists

From version 7.0.2 onwards, **Dynamic List** is supported as a '**Custom**' input type in both the Manual Trigger step and the Manual Input step. A 'Dynamic list' is a list with dynamic values that is set using a playbook, i.e., the options of the list are defined using JSON or comma-separated values, set using Jinja and gets displayed as a list in a user input prompt. You could use dynamic lists in cases such as:

- **Independent Playbooks**: You might need to create a list for the manual trigger or manual input that can be automatically included as part with the exported playbook, i.e., the playbook step holds the logic of the items and therefore does not require a custom picklist to be exported.

- **Unauthenticated Picklists**: Picklists cannot be loaded in case of unauthenticated inputs. In such cases, pre-defined JSON lists can be used to present multiple options to users for their selection.

- **Constantly Changing Picklists**: There could be picklists such as picklists based on MITRE threat hunting categories, which are ever changing. In such cases you can prompt users to categorize the threat based on the

loaded list of categories. Then, based on the selected threat, you can automatically show the sub list for selecting techniques.

- **Filtered Record List**: You can present a filtered lists of records in a manual input to users for selection. The filter would vary depending on logic in the playbook.

For example, if you want to a MITRE techniques dynamic list, in which you can define the various MITRE tactics and then based on the users' selection, display the various techniques associated with the tactic. To add such a dynamic list to a User Prompt of a manual trigger or manual input, do the following:

1. In the `User Prompt` section, click the down arrow, and then click **Add Field**.
2. Select the **Input Type** as **Custom** and from the **Field Type** drop-down list, select **Dynamic List**.
3. If you want users to select an options from a list of options, then using the **List** mode, you can add various options to the dynamic list using comma-separated values, i.e., add the various options in the **Options** box in `List Options` section. However, if you want to present the users with multi-level options, as per our example, then you need to use the **JSON** mode. To display multi-level options, use an object with a string `"value"` and list `"options"` for sub-

options as shown in the following image:

4. In the **Field Label** field, add an appropriate title for the user prompt, such as `Choose the Mitre Technique`, then click **Save** to save your changes to the step, and then click **Save Playbook**.

When you run the playbook on a record, you will see that the Dynamic list will appear in the User prompt as shown in the following image:
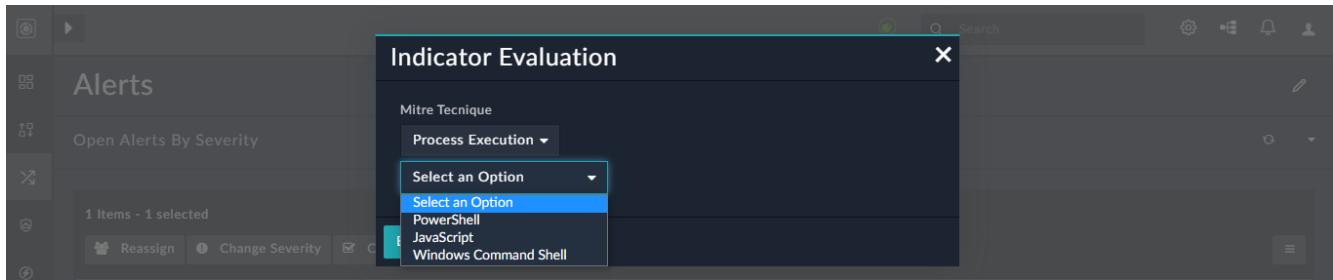
User Input Prompt - Custom date/time field usage notes

When you add a custom **Date/Time** field as an input parameter in an `Input Prompt`, then that Date/Time appears correctly in the Input Prompt, however any create record or update record that uses this custom date/time field will display the created/updated record as **01/01/1970**.

For example, in a Manual Trigger set on the `Alerts` module, when you add a custom **Due Date** field of type `Date/Time Field`, whose due date is set as **Current Date +1 Day**, and the step following the Manual Trigger step is a Create/Update Record step to create/update an alert record that uses the custom due date, then the alert record get created/updated with the Due Date set as 01/01/1970. This happens since the create/update record step requires the date/time in the `epoch` time, which is not the format in which currently the create record step is receiving the date/time. To fix this, in the create record step, in the **Due Date** field, add the following: `{{arrow.get (vars.input.params.dueDate.int_timestamp}}`. The `{{arrow.get(jinja varibale).int_ timestamp}}` converts the value of the date/time field into the epoch date/time.

Triggers

Trigger Data

Within the context of dynamic variables, the trigger step allows access to all data within the inbound transaction using the Dynamic Value prefix within the Jinja2 template formatting, for example, `{{vars}}`.

> See the Dynamic Variables and Dynamic Values chapters for more information on using Dynamic Variables within a Playbook environment.

Standard information that is packaged includes, but is not limited to, the following:

Key	Information Type	Applies To
auth_info	This displays the type of authentication invoked by the user who triggered the playbook. It can be no authentication, basic authentication, or CS HMAC authentication.	All
currentUser	The IRI of the current user who triggered the playbook.	All
last_run_at	The last date of execution for a playbook that is run on a periodic basis.	Scheduled
request.base_uri	The root of the host URI on which the playbook is executing, for example, `https://fortisoar.sampleurl.com`	All

`request.uri`	The full URI route of the API endpoint used to invoke the playbook.	All
`input.records`	An array of records under the operation. For post-create, post-update, and manual triggers that have a single records, the array contains only one record that can be accessed using `input.records[0]`.	Manual trigger, Post-Create, and Post-Update triggers
`input.params['api_body']`	The payload of the request in case of the custom API endpoint trigger.	`API` trigger only
`input.params.<param_name>`	Inputs that are specified using the `Input Parameters` option in a playbook.	All
`request.headers`	All the headers sent with the request that invoked the playbook.	All
`request.headers['X-RUNBYUSER']`	The IRI of the current user who triggered the playbook.	All
`previous`	Specific to the `Update` trigger. It shows the original version of the record data before being changed.	`Update` trigger only
`resource`	The module targeted by the playbook.	Database triggers
`request`	The full request object that initiated the playbook.	All
`request.data`	The cleaned data, if in JSON format, associated with the `request.body`.	All
`request.method`	The RESTful method by which the playbook was triggered, only `POST` or `PUT`.	All

Database Triggers (On Create, On Update, and On Delete)

In the case of a database trigger, such as On Create, the record which triggered the playbook is included within the API request and is accessible. The format of the record data will be identical to the format accessible within the standard Module endpoint for that record type.

Sample Data

Standard keys for data available within the `vars.input.records[0]` includes the following when it comes from an internal trigger, such as a On Create. The `%` indicates a placeholder for data that would be present in a real request in the general format.

```
{
    "auth_info": {
        "auth_method": "CS HMAC"
        },
    "currentUser": "%CURRENT_USER%",
    "last_run_at": null,
        {
        "input": {
        "records": []
```

```
                },
        "request": {
        "method": "PUT",
        "body": "%RAW DATA INCLUDED IN THE BODY OF THE REQUEST",
        "query": [],
        "data": {
                "%CLEANED DATA OBJECT FOR RECORD IF IN JSON%"
                },
        "baseUri": "https://fortisoar.sampleurl.com",
        "uri": "https://fortisoar.sampleurl.com/api/3/%MODULE%/%UUID%",
        "headers": {
                "connection": "keep-alive",
                "x-php-ob-level": 1,
                "origin": "https://forisoar.sampleurl.com",
                "authorization": "Bearer %token%",
                "user-agent": "%AGENT%",
                "cookie": "%COOKIE%",
                "accept": "application/json, text/plain, */*",
                "content-length": "%%",
                "referer": "https://fortisoar.sampleurl.com/modules/%MODULE%/%UUID%",
                "content-type": "application/json;charset=UTF-8",
                "accept-encoding": "gzip, deflate, sdch, br",
                "host": "fortisoar.sampleurl.com",
                "accept-language": "en-US,en;q=0.8"
                }
        },
        "previous": {
                "data": {
                "%DATA OBJECT FOR PRIOR RECORD%"
                }
        },
        "resource": "%MODULE%",
}
```

As part of consolidating inputs for various types of triggers, all request parameters for all the different types of triggers have been consolidated under `vars.input`. For On Create, On Update, or Manual triggers, the record details are available under `{{ vars.input.records }}`. For API triggers, the request data is available under `{{ vars.input.params['api_body'] }}`. To avoid data duplication, `vars.request.data` is being deprecated in FortiSOAR 6.0.0 and it is recommended to use the above equivalents under `{{ vars.input }}`.

Manual Triggers

The Manual trigger payloads have a similar structure to the database triggers, and the payloads of both manual and database triggers are accessible using `vars.input.records`. The `records` array is an array of JSON objects, one object for each record that was passed in as a part of the request.

For instance, if you click the **Execute** button on the grid by selecting five record checkboxes, the data from all five records will be included in the `records` array in their raw format.

The Manual Trigger step also provides you with options to specify whether the action must be Executed **Once** or **For Each Record**. This enhancement makes it more effective to handle multiple selections since now you do not require to write two playbooks and map the second playbook in the first playbook. For more information, see Manual Trigger.

Custom API Endpoint Triggers

Internal triggers will always have a JSON format, but Custom API Endpoint triggers are initiated from external systems and might not always come in JSON format. Currently, custom Custom API Endpoint triggers can accept any format of the inbound body data, but this data might not be accessible within the environment in a structured way.

As an example, an XML request is not available in the environment until it has been parsed by a separate step. This can be done any time after the trigger step but must be done before referencing any variables that would be expected out of the XML structure.

XML can have a more sophisticated data structure than JSON and therefore, might require custom parsing for correct handling of XML data. A custom parsing step to convert XML to a dictionary format is present in the Utilities connector, "Convert XML to Dictionary".

Referenced Trigger

The Referenced Trigger step will always be called from another playbook. Therefore, it can get the environment using `input.params.<param_name>`.

Bear in mind that chaining multiple playbooks can overwrite the variables in your environment, such as the `request` object. Use the `Set Variable` step to give unique names to prevent this from happening. You can use the `Set Variable` step, to create an input parameter with a unique name that will be available in the parent (calling) playbook. To add an input parameter, in the playbook designer, click the **Tools** menu and select **Edit Parameters**.

Data Inheritance

See the References section to understand how data inheritance works in FortiSOAR.

Playbook Steps

At the core of Playbooks are Steps. Steps represent discrete elements of data processing during the course of the Playbook.

`People`, `System Assigned Queues`, and `Approval` modules are removed from playbook steps since these are system modules and used for administration purposes.

Steps can be linked together in sequences to determine the flow of the Playbook, starting from the Trigger.

The Playbook Designer displays Playbook Steps only after you have added a Playbook Trigger.

Use dynamic values or variables in playbooks to access values of objects or perform lookups. Dynamic values can be passed to playbook steps as arguments directly, or they may be embedded in a larger string, where they will act more as global variables, getting replaced by a string representation of themselves. For more information, see the Dynamic Variables chapter. You can also use Dynamic Values to generate jinja templates, which can dynamically define various conditions within steps in a playbook. For more information, see the Dynamic Values chapter.

In case of any playbook step, if the input value for any field is in the JSON format, then you must enter the data in single quotes for example, `'{"company":"fortinet"}'`.

To update a picklist using a playbook, you can directly add the jinja for the picklist in the `{{"picklist name"|picklist("itemvalue of picklist")}}` format, for example, `{{"AlertStatus"|picklist ("Open")}}`. The IRI Lookup option in the Dynamic Values dialog also allows you to select a picklist. For more information, see the Dynamic Values chapter.

Click the **Add Custom Expression** ( ) button to toggle fields such as, **Date/Time**, **Rich Text**, **File Selector**, **Picklist**, **Lookup**, and **Checkbox** fields and add custom (jinja) expressions to these fields. Ability to add jinja expressions to these fields enables you to write advanced playbooks. Once you click the **Add Custom Expression** button, you can also use the **Dynamic Values** dialog to add expressions to these fields. For more information on the **Dynamic Values** dialog, see the Dynamic Values chapter.
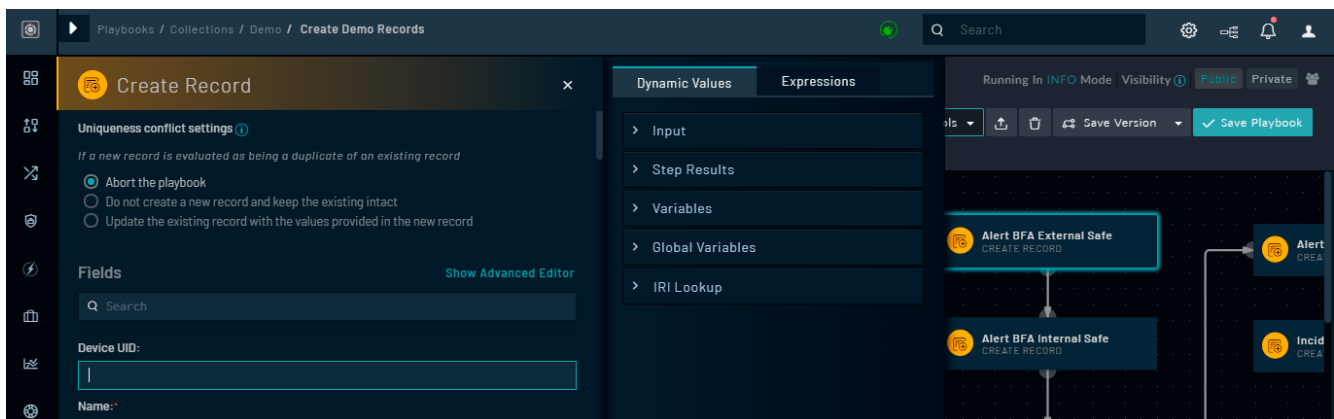
Once you have saved the step, a graphic representing the step displays on the designed canvas in the upper left corner. You can create a link between the trigger and the step, for more information, see the Introduction to Playbooks chapter.

Double-clicking on the step reopens it and allows the user to edit the step or delete the step entirely by clicking **Delete Step**.

You can add variables, loops, conditions, and custom messages directly in the playbook step itself, and they get added in the `Step Utilities` section. You can also add a sample output (mock output) for cases where you do not want to execute a step but mock the output so that the playbook can move forward. You can also click the **Yes**/**No** button beside the **Ignore Error** checkbox to allow the playbook to continue executing even if the playbook step fails. These actions that you can use to extend a playbook step are present in the footer of the playbook step as shown in the following image:
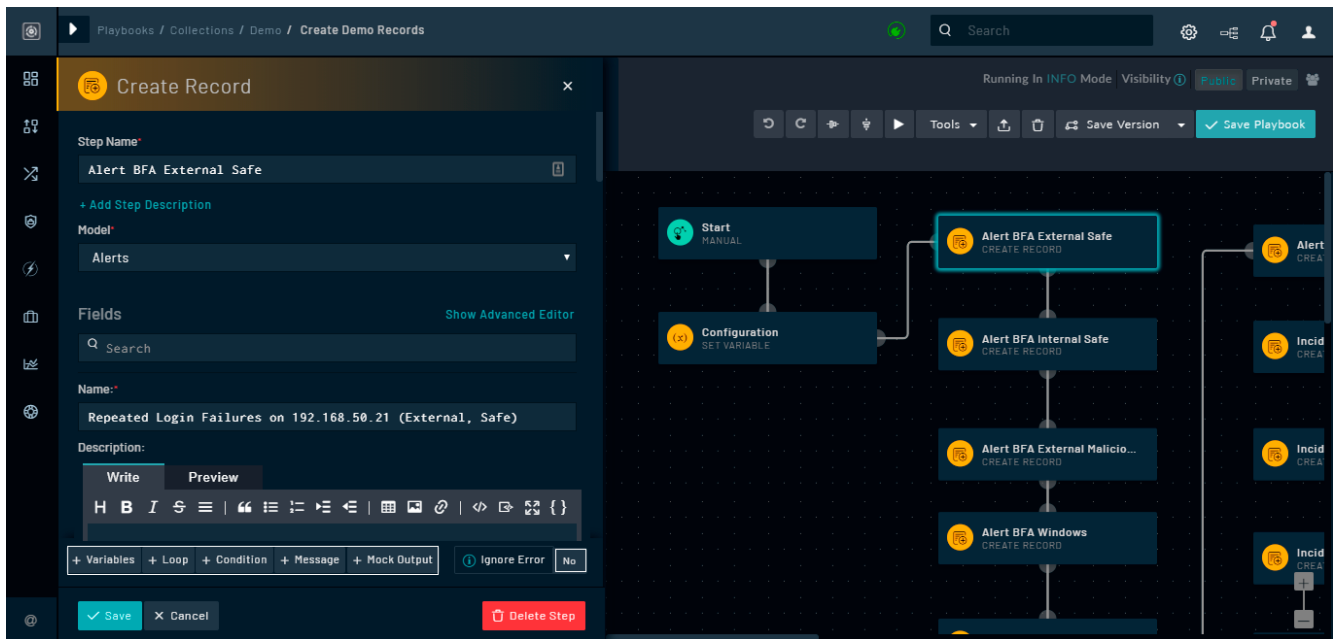
Playbook actions used for extending playbook steps

Condition

To add a condition to a step, click the **Condition** link that is present in the footer of the playbook step. Clicking the **Condition** link adds the **When** textbox, in which you add the expression (condition) based on which the decision to execute the playbook step is taken. If the condition is met, then the playbook step is executed. If the condition is not met, then the playbook step is skipped.

If you use `when` without the `for each` loop, then it applies to the step level and determines whether the playbook step will be executed or not and it is the first thing that is evaluated for the step. If you use `when` with the `for each` loop, then it applies inside the for loop for each item.
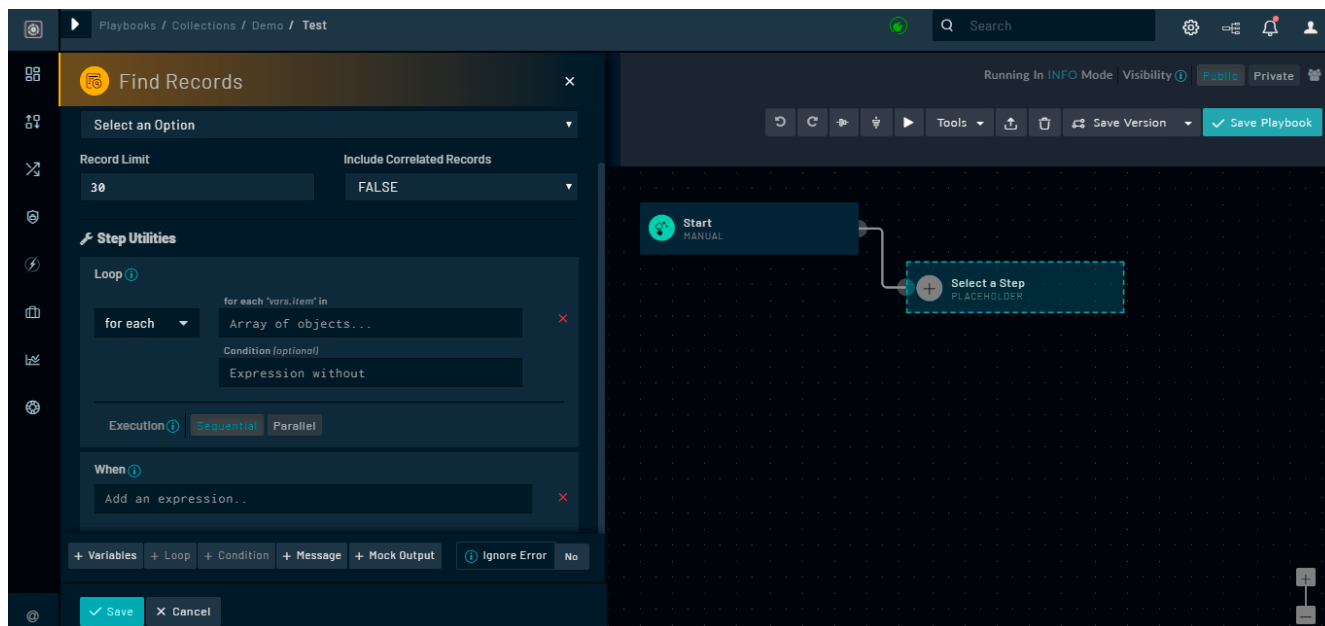
Variables

To add a variable to a step, click the **Variables** link that is present in the footer of the playbook step or add the variable in the `Variables` section of the step. Using **Variables** you can store the output of the step directly in the step itself. Therefore, instead of having to use the **Set Variable** step frequently within a playbook to collect specific response data and provide a contextual name to the output, you can use **Variables** in the step itself. You can also store custom expressions in variables, which can be accessed within the playbook.

> ⚠️ Do not use the following reserved words as a variable name in Variables: `for_each`, `do_until`, `ignore_errors`, `condition`, `message`, and `mock_result`.
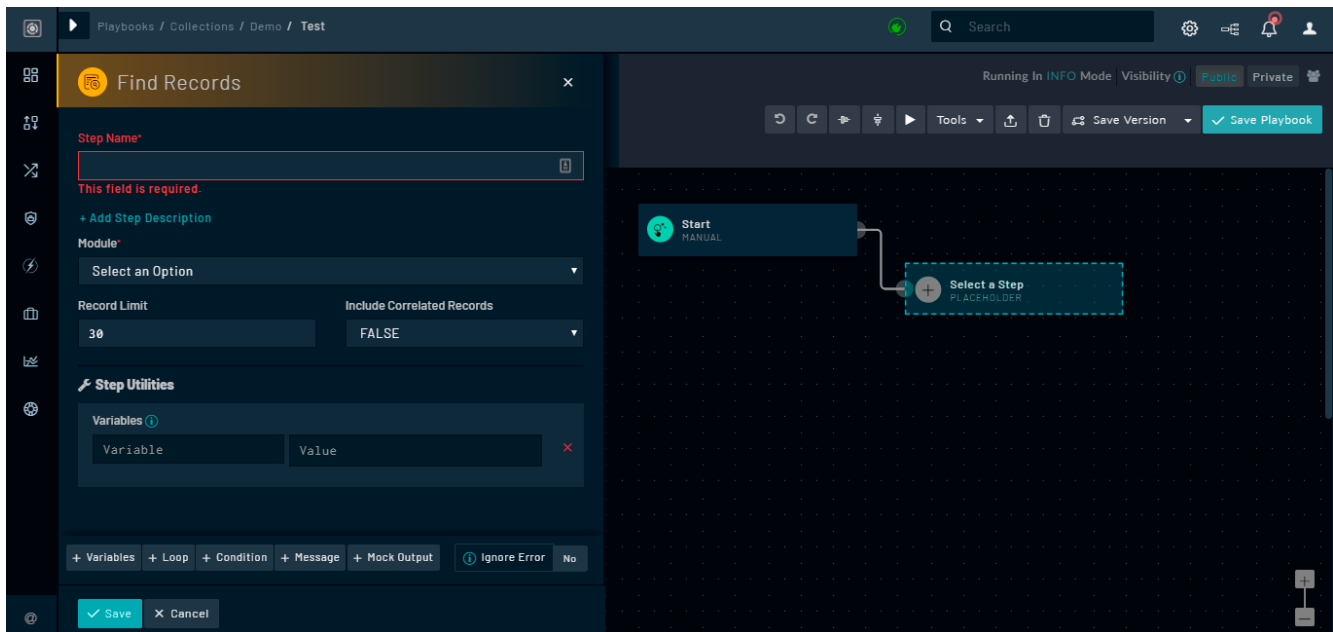
Use Dynamic Values to add or store the output of the current step directly in the step itself as shown in the following image:
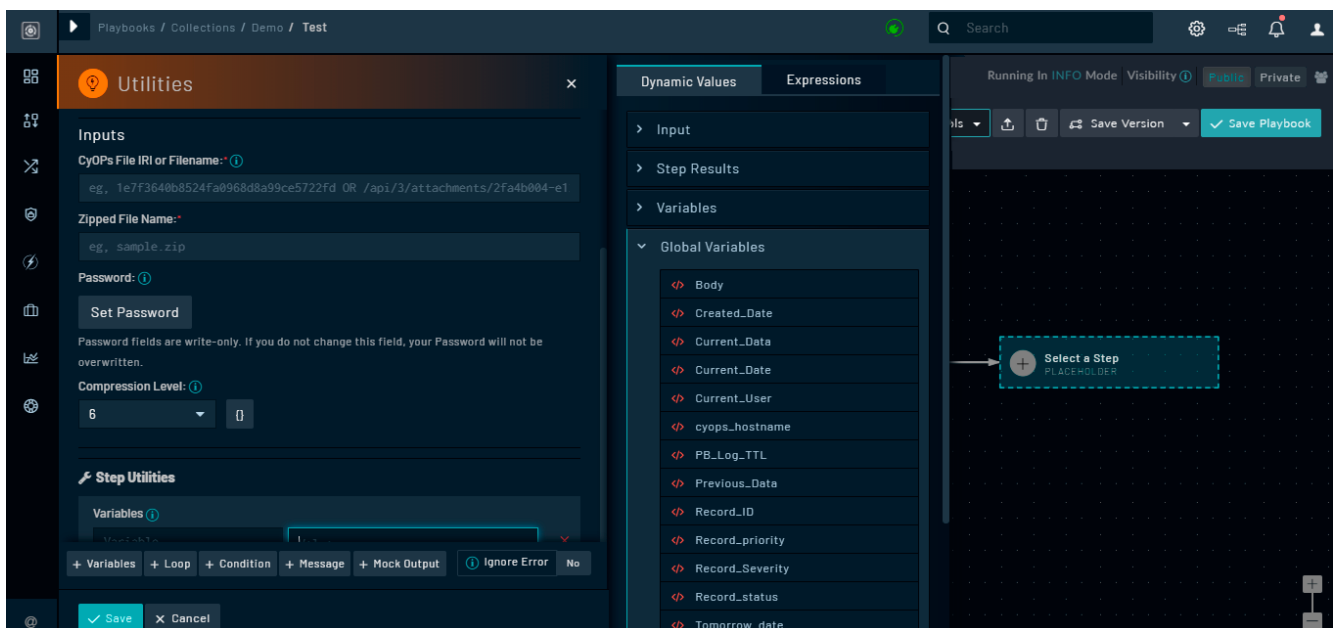
For more information on Dynamic Values, see the Dynamic Values chapter.

Loop

To iterate the playbook step, click the **Loop** link that is present in the footer of the playbook step. There are two types of loops that you can add to a playbook step: the for each loop and the do until loop.

The **for each** loop can be added only once in a playbook step. The input for the `for each` loop is an array of objects and the `for each` loop iterates for the length of the array. To access the object of an array use the reserved keyword

item. An example of an array of alerts objects is `[{"name":"Alert Name1"},{"name":"Alert Nam2"},` `{"name":"Alert Name3"}]` and to access an object of an array, use `vars.item.name`. You can optionally add a condition to the `for each` loop, based on which the loop will be executed.

The Loop option has three modes: Bulk, Sequential, or Parallel.
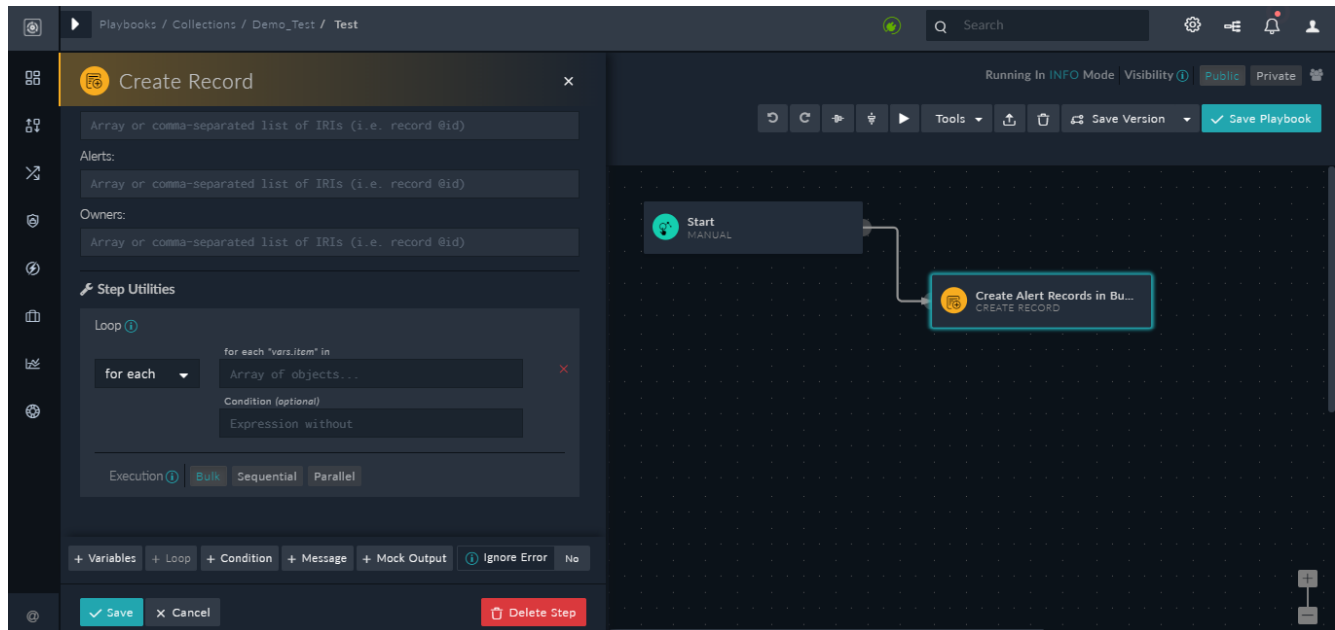
The Bulk mode creates all records in a single API request and is the most optimal and recommended method of creating or upserting records in bulk. This is also the default mode when you add a new "Create Record" or "Update Record" step in a loop. If you are inserting larger number of records that causes the API call to time out, then you can insert records in batches. For more information, see the Batching large datasets when using the 'Bulk' option section.

The Sequential modes sends the API records separately for individual records, and one after another. So, the playbook step can abort at the first failure, without proceeding to create further records. The Parallel modes sends separate API requests for each record creation but using multiple threads to do so.

You can choose whether you want to execute the playbook step in parallel or in a sequence for the given items. Sequential execution of the loop works on one item at a time in a serial manner, whereas parallel execution utilizes multiple parallel threads to work on the items, resulting in better performance. You can choose your option using the **Execution** toggle as shown in the following image:
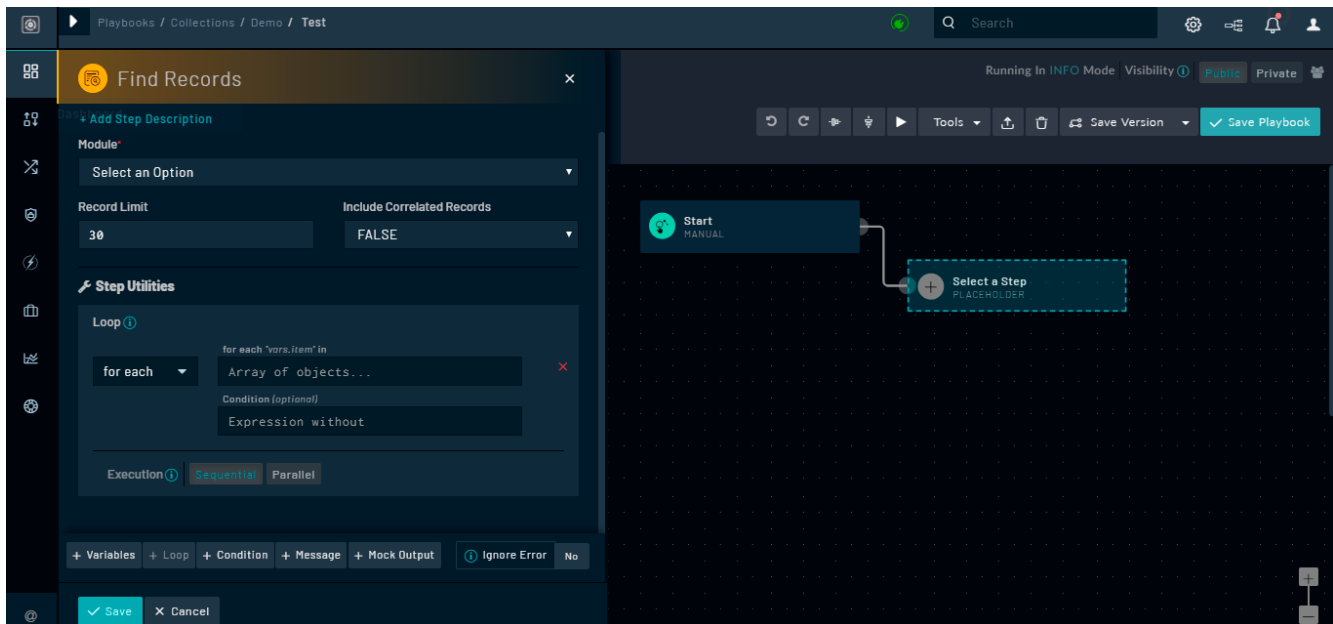
The workflow engine can execute multiple independent paths in parallel threads. Parallel branch execution means that two or more paths execute the independent paths in parallel. This enhancement is transparent to the end-user, but in some cases, this could lead to a change in the behavior of certain playbooks compared to the old sequential behavior as the step execution order might change. If any of your existing playbooks fail due to a previous step result not found, or similar reasons, you can run a test to find out the cause of the failure by turning off the parallel execution feature.

You can enable or disable parallel execution by changing the value (true/false) of the `PARALLEL_PATH` variable in the `[Application]` section in the `/opt/cyops-workflow/sealab/sealab/config.ini` file. By default, the `PARALLEL_PATH` variable set as `true`.

You can also tune the thread pool size and other settings for parallel execution. For more information about settings that you can set for optimizing your playbooks, see the Debugging and Optimizing Playbooks chapter.

You can also execute the referenced playbook asynchronously from the parent playbook. In this case, the parent playbook continues to execute the remaining workflow, without waiting for the referenced playbook to finish.
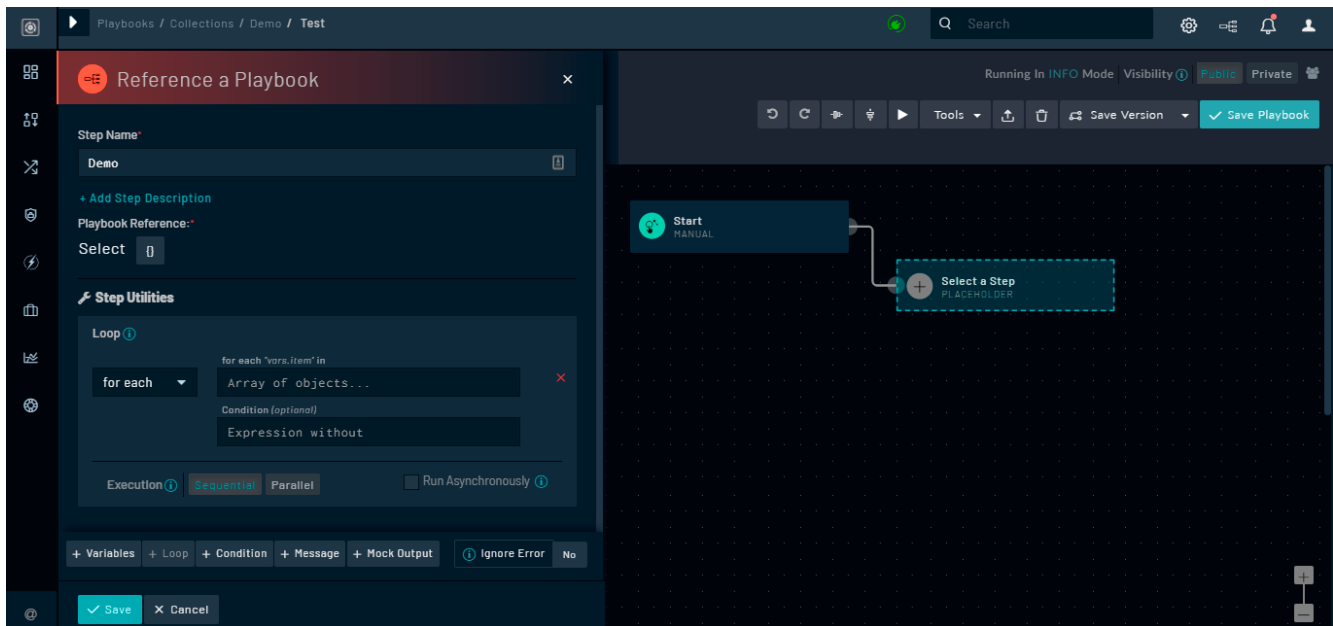
Click the **Run Asynchronously** checkbox to enable the referenced playbook to run asynchronously.

If you select a child playbook to be executed as asynchronously, then you will be unable to use the output of the child playbook in the parent playbook. Therefore, you must be cautious while using asynchronous mode, and should only use this mode when you want to execute child playbooks independently. For example, in the case where you want to ingest the records and not perform any action on the output.

The **do until** loop will execute the step at least one time and will continue to run until the condition specified is met, or the number of retries is reached. You can configure the number of retries the playbook step will execute to meet the condition and also the delay in seconds before the step gets re-executed in a loop. By default, the number of retries is set to `3` and delay is set to `5` seconds.
 In a do until loop, you can access the result of the current step with the `vars.result` notation. For example, to keep trying to run a connector action until it is successful, you can set the condition to `vars.result.message == 'Success'`. You would also need to check the **Ignore Errors** box to ensure the playbook does not stop if that step fails.

Do not use `do until` with `when` or `for_each`.
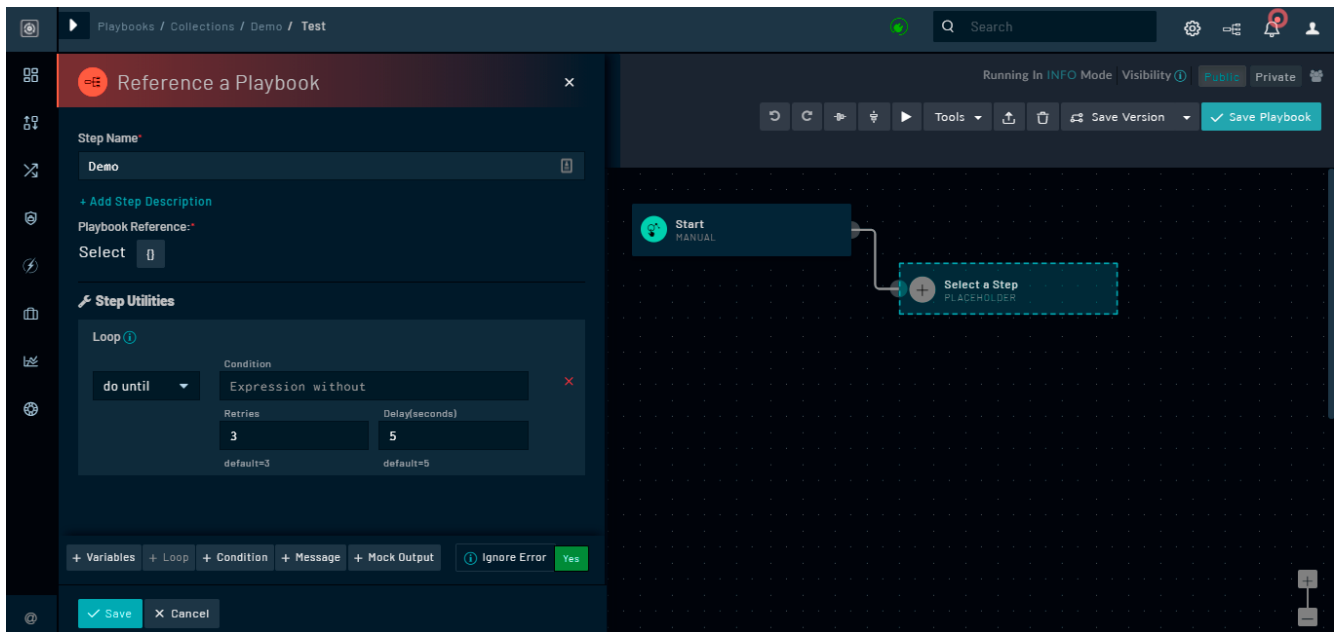
Message

You can add a custom message for each playbook step to describe its behavior. You can also use Dynamic Values to add jinja values to the messages. Dynamic Values also displays the output of the current step in the **Message** step. For more information on Dynamic Values, see the Dynamic Values chapter.

These messages appear in playbook logs and are also displayed as part of the collaboration panel. The Message content can be rendered in HTML or Markdown, depending on the whether you have set the **Contents** field in the "Comments" module to Rich Text (Markdown), which is the default or as Rich Text (HTML). The following image displays the Message content in the default Markdown editor:
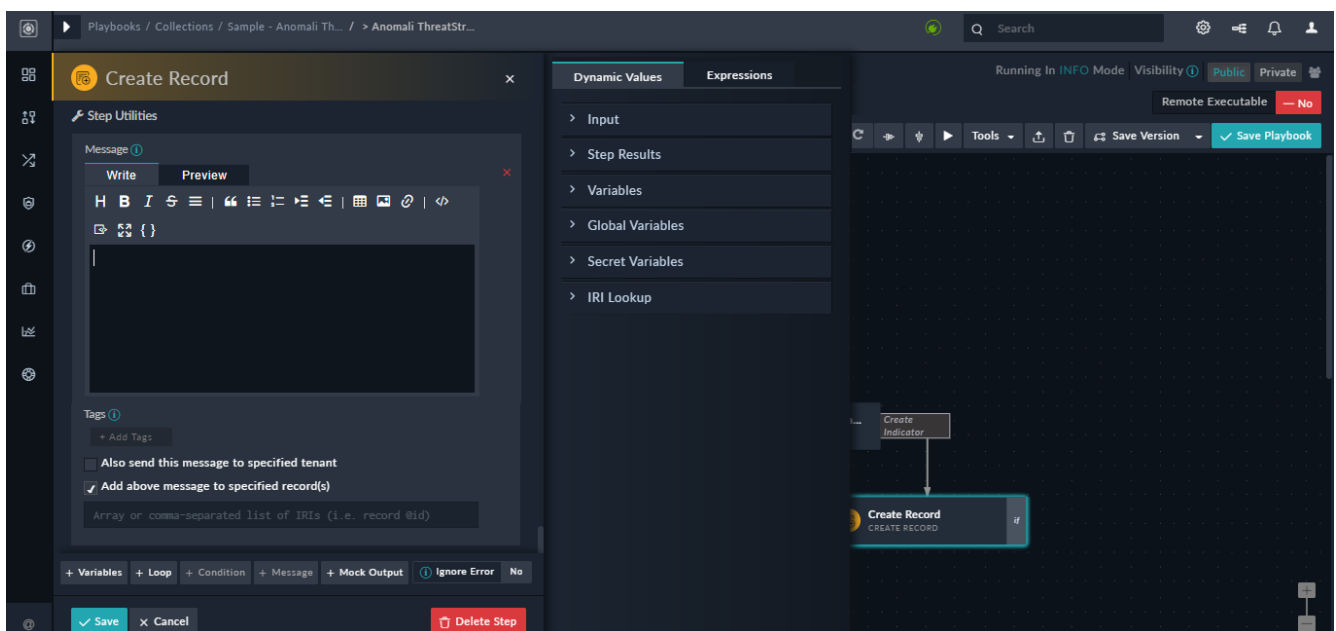
You can also add the custom message to another record(s), by clicking the **Add above message to specified record (s)** check box. If the **Add above message to specified record(s)** check box is selected and you have provided the IRI of the record(s) to whose collaboration panel the message requires to be added, then the message is added to collaboration panel of those record(s). If the **Add above message to specified record(s)** check box is selected and you have not provided the IRI of the record(s) to whose collaboration panel the message requires to be added, or you have not selected this check box, then the message is added to collaboration panel of record that triggered the playbook.

In case of multi-tenant configurations, if a playbook that contains steps with "Messages" is added to the record that triggers the playbook on the master node, you can choose to replicate the comments that are linked to the record on the tenant node, so that a user on the tenant node can follow the investigation that is being conducted on the record. To replicate comments on the tenant node, click the **Also send this message to specified tenant** checkbox, and from the **Select Tenant** drop-down list, select the tenant node on which you want to replicate the comments or click **{}** to specify tenant IRIs in this field.

Mock Output

You can mock a step output in cases where you do not want to execute the playbook step but ensure that the playbook can move forward using the mock output. This can be useful when you want to debug playbooks. You can also use Dynamic Values in the Mock Output step.
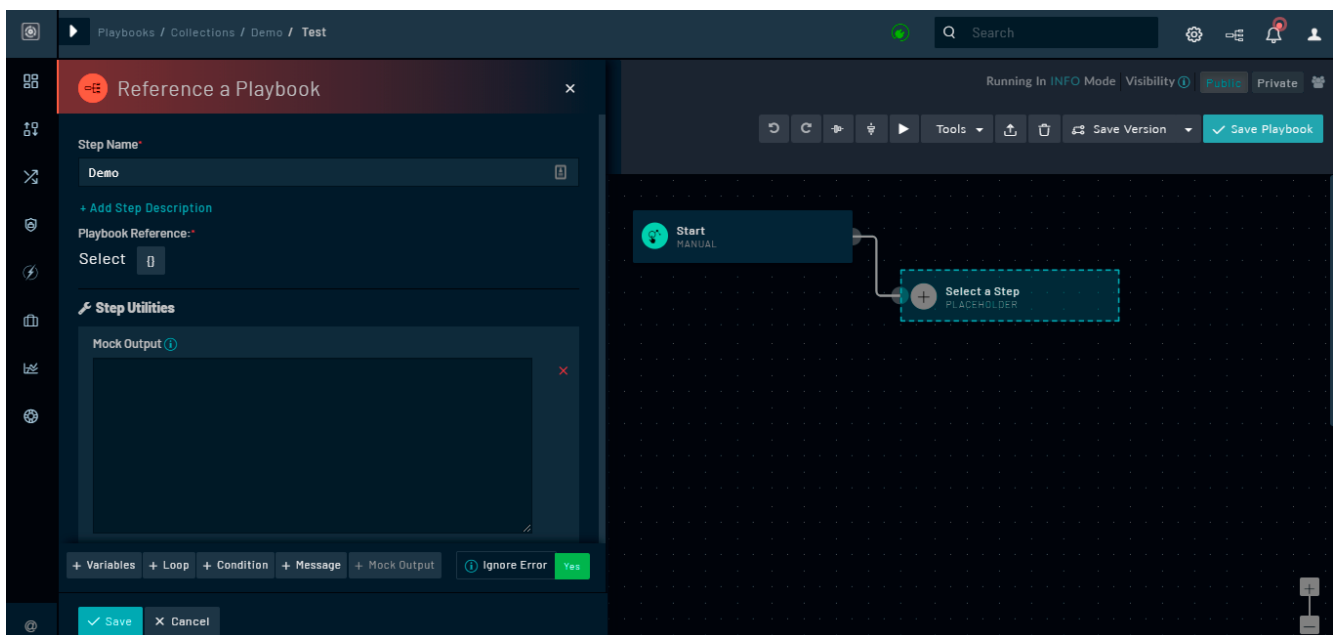
If you want to use *mock* output for your playbook steps, then you must add a variable named 'useMockOutput' and set its value to 'true,' using the Variables option in the trigger step. If you do not declare this variable or set the value of this variable to 'false,' then the playbook will use the actual step outputs for execution. Also, ensure that you write `useMockOutput` precisely as is since this variable name is case-sensitive. For more information on variable, see Variables.

Ignore Error

You can click the **Yes**/**No** button besides **Ignore Error** to allow the playbook to continue executing even if the playbook step fails.
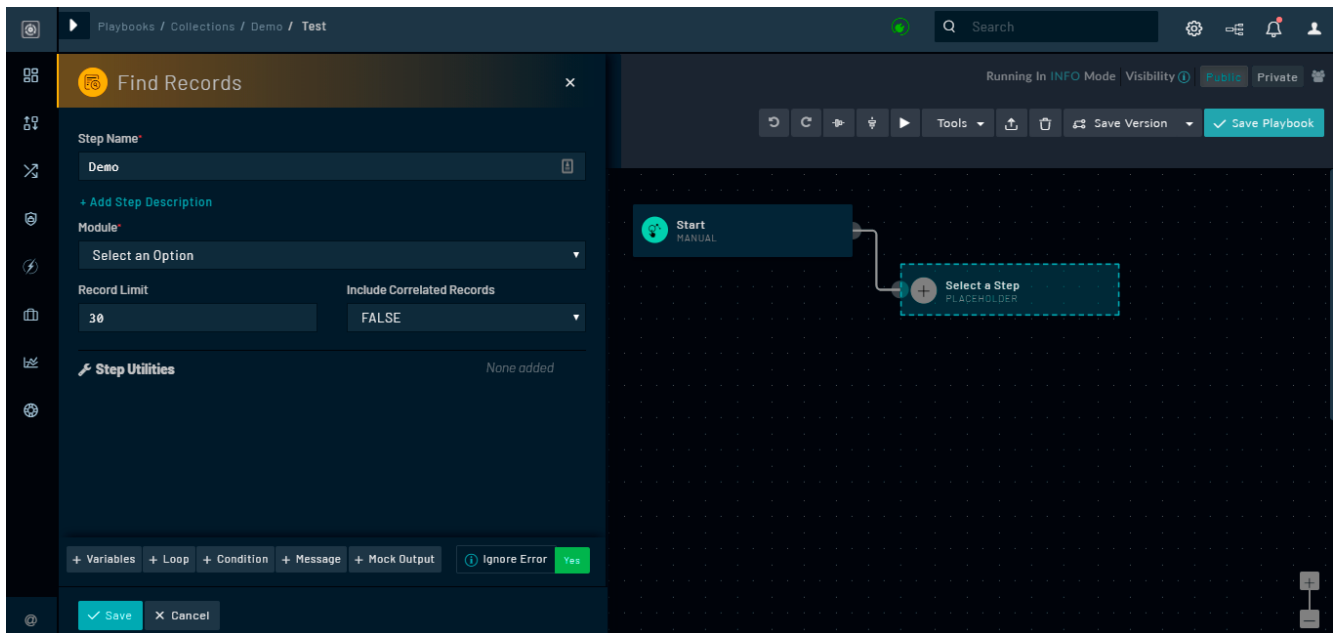
However, in the playbook log, the status of this step will be `Finished with Error`. Open the playbook log by clicking the **Executed Playbook Logs** icon ( ) that appears on the top-right corner of the FortiSOAR screen. Click the step whose log you want to view, and in the `Step View` section, the status of the playbook is displayed in the `status` item, and the error is described in the `result` item.

The following sections explain the various steps used in playbooks.

Core

Create Record

Use the **Create Record** step to create almost any record type in the system. All required fields must be entered to match the model metadata for that specific record type. To create a record, select the module in which you want to create the record from the **Model** drop-down menu, which displays the `Create Record` form (**Form Editor**). Note that the fields displayed are specific to the entity type selected, and any conditional data requirements will be activated the same way as if the record was being added using the entity's model itself.

If the data entity needs to reflect data specific to the entity that triggered the playbook, then use *dynamic values* in the fields.

> To set the name of the incident name of the triggering entity, put the following in the Name field: `{{vars.input.records[0].name}}`.

Module editor supports the "JSON" field type. You can also convert data of a field of type `text` to JSON, using the `toJSON` Jinja filter. For example, `{{ vars.result.data | toJSON }}`.

If the fields of the record being entered will always have the same data, enter the text in the corresponding fields and click **Save**.

If in the Create Record step, you are specifying any `Date/Time` field in the jinja format, then that date/time field must be in the `epoch` format. To convert the input date/time field to the epoch time, you can either add the following Jinja value: `{{arrow.get(jinja variable).timestamp}}` or use the DateTime Expression library to enter the data directly in the JSON format by clicking the **Add Custom Expression** (▯) button button. Clicking **Add Custom Expression** (▯) button displays `Dynamic Values`, which displays the fields that you can directly edit either in the format of an attribute map (**Tree** view) or code (**Code** View).

Important: In version 7.0.0, FortiSOAR has updated the arrow library due to which the `timestamp` attribute has been changed into `int_timestamp` for *DateTime* jinja expressions. New playbooks must use the `int_timestamp` for any *DateTime* jinja expressions. For more information see the Dynamic Variables chapter.

You can also specify the date by clicking the **Select Date** link, which displays the Calendar from which you can choose the date/time.

From version 7.0.0 onwards, in case of the 'Create Record' and 'Update Record' steps, if your administrator has enabled any 'Lookup' or 'Picklist' type of field to accept the values generated from the recommendation engine, then you will see an **Auto populate** checkbox appearing beside this field.

To auto populate values in related (many-to-many) fields, in the 'Create Record' and 'Update Record' steps, ensure that the input provided in the playbook is the 'array' (list) format, and not 'comma-separated values'. For example, if you want the **Indicators** multi select picklist to be auto populated with common indicators in alerts records, then you must ensure that the indicators list that you provide in the playbook that contains the 'Create Record' or 'Update Record' step is in the 'array' (list) format.

A example of a field that has been enabled for recommendation is the 'Type' field in the following image:

If you select the **Auto populate** checkbox, and users have not specified any values for such fields, then the value of such fields get auto-populated with the values from the recommendation engine that is based on learning from past similar records.

It is possible to relate records with any valid relationships in the system. You can link the record that you are creating to a record in a related module. The **Create Record** step now displays a list of modules, in the `Correlations` field to which you can link the record that you are creating. For example, if you want to create an alert and therefore you have selected **Alerts** from the **Model** drop-down menu, the `Create Record` form will display related linking module fields, such as `Incidents`, `Indicators`, `Assets`, and `Attachments`. The Create Record step (for the upsert cases) and the Update Record step, the `Correlations` field, displays the records that are already linked to the created record. You can choose to overwrite the older relationships that are added to the created record, by clicking the **Overwrite** option in the `Correlations` field or append the new relationships to the relationships that are already added to the created record, by clicking the **Append** option in the `Corrleations` field.

To link the newly created record, in the linking module field, add the IRI of the record to which you want to link the newly created record or add the respective jinja values. You can link multiple records using multiple comma-separated IRIs. For example, to link an alert that you are creating to an incident record, select **Alerts** from the **Model** drop-down menu and in the `Incidents` linking module field, add the IRI of the incident record, such as `/api/3/incidents/9a1142d2-adbf-4faf-a477-d8ff54419808` or add the jinja value or the incident record, such as: `{{vars.input.records[0]['@id']}}`. You can also use the array format to specify the IRI, `["/api/3/incidents/9a1142d2-adbf-4faf-a477-d8ff54419808"]`, or also add the jinja value in the array format, `["{{vars.input.data.records[0]['@id']}}"]`. To get the IRI for a record by navigating to the related module (`Incidents` in our example), for example, **Incident Response** > **Incidents** and select the record that you want to link. In the `address` bar, you will see the complete URL for that record. For example, https://{{*Your_FortiSOAR_IP*}}/modules/view-panel/incidents{{*UUID*}}

https://{{*Your_FortiSOAR_IP*}}/modules/view-panel/incidents/9a1142d2-adbf-4faf-a477-d8ff54419808.

It is recommended that you do not link more than 99 records in a single call. If you need to link more than 99 records, then run the update step in a loop with batches of 99 records.

The behavior of linking records relationships has changed in version 7.0.0 because if there is a record that is linked to thousands of other records, an update to such records causes constant high CPU usage. An example of such a record would be indicators like org name that get extracted as part of every alert and get linked to thousands of alerts. Therefore, it is recommended that you **link a maximum of 99 records in a single call**. This is because, if there are less than 99 records linked then the framework checks if the record being linked is already present in existing relations and if the same record is linked again and again, post-update triggers on relation "isChanged" is not triggered, also the linking is not audited again every time. However, from the 100th linked record, the framework only looks at the `__link`, `__unlink` keys, and hence, if the same record is linked again and again, post-update triggers on relation "isChanged" will get triggered, and also the linking gets audited again every time.

When you are creating a record using playbook you can also enforce record uniqueness by defining unique constraints on the records of a module. For information on how to define record uniqueness using the Module Editor, see the *Application Editor* chapter in the "Administration Guide."

For modules that have unique constraints defined, then the option that you choose in the `Unique conflict settings` section will determine the behavior of the playbook, which are as follows:

- **Abort the playbook**: This is the default behavior. The playbook will fail if a duplicate record is found.
- **Do not create a new record and keep the existing intact**: The playbook does not make any changes to the existing record and the existing record is returned *as is* as a result of execution of this step. The subsequent steps of the playbook will work on the existing record if they refer to this step result.
- **Update the existing record with the values provided in the new record**: The playbook updates the existing record with the new values that you have specified in this step.
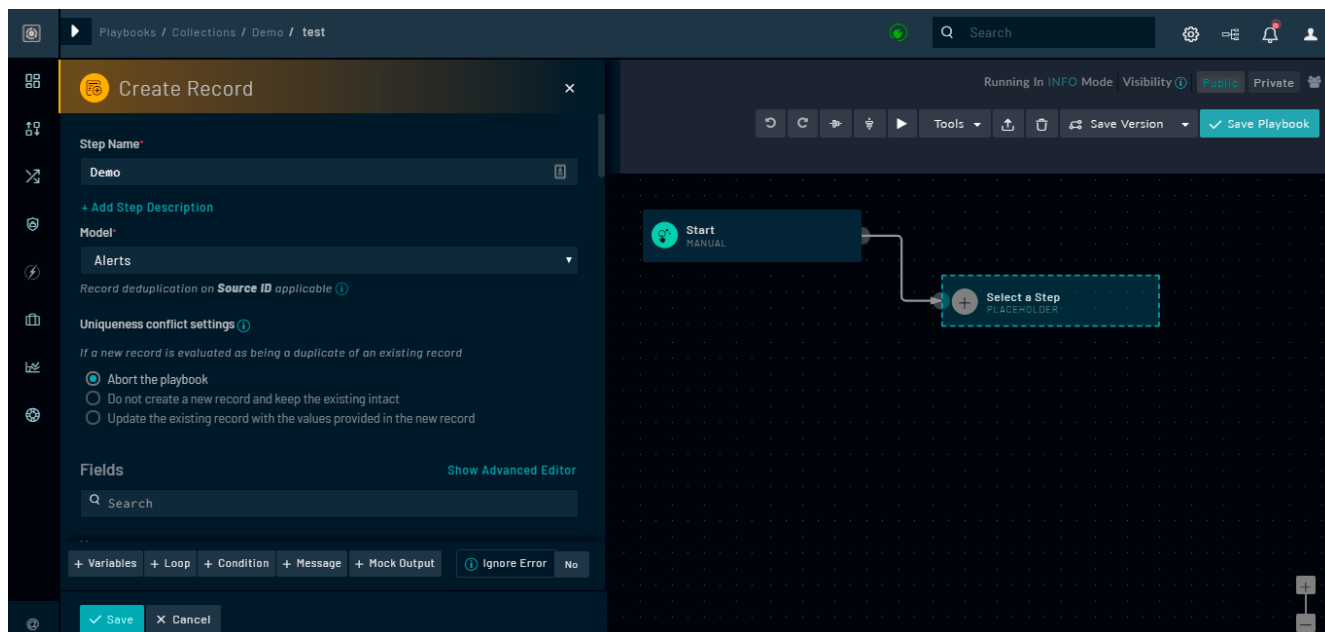
For example, if *Source ID* is specified as a unique constraint on an "Alert" module (as shown in the above image) then you cannot create a record having the same source ID. However, if you have selected the **Update the existing record with the values provided in the new record** in the `Uniqueness conflict settings` section, then the existing record will be replaced with the updated values.

Note: If you have imported playbooks into your FortiSOAR system or have upgraded your FortiSOAR system, and you have playbooks that contain the Create Record step with the Upsert option, i.e., you have selected the **Update the existing record with the values provided in the new record** option, then such playbooks will delete the values of all fields (except for the ones that are updated) in the existing records. To avoid the deleting of field values, open the **Create Record** step for all such existing playbooks and save the step and then save the playbook.

Upsert behavior for uniqueness will not work for fields that are marked as encrypted.

You can add tags in the **Create Record** and **Update Record** steps. You can add tags to the record that you are creating using the `Tags` field. Special characters and spaces are also supported in tags; however, the following special characters are not supported in tags: `'`, `,`, `,`, `"`, `#`, `?`, and `/`. Tags are useful in searching and filtering records. When you are updating a record, the `Tags` field, displays the tags that are already added to the created record. In the Create Record step (for the upsert cases) and the Update Record step, you can choose to overwrite the older tags that are added to the created record, by clicking the **Overwrite** option in the `Tags` field or append the new tags to the tags that are already added to the created record, by clicking the **Append** option in the `Tags` field.

Once you create the **Create Record** step, the playbook is automatically prompted to create a data record as specified in the step with either specific static text or record-relevant data using dynamic values.

When a record is created from a playbook, then that record's ownership includes the teams that are part of the "Playbook Appliance" including the admin team (SOC team). So, the record will be visible to all members of the teams that are part of the "Playbook Appliance", and their siblings and parents in the team hierarchy. If you want to change the ownership of the record, in the playbook, after the step to insert the record, add the immediate next step that will assign the desired team or user as the owner of the record.

Create or Upsert Records in Bulk

You can also create or upsert records in bulk by using the **Bulk** option in the `for each loop` for "Create Records" and "Update Records." To create multiple records in a single request, for example, while ingesting from a data source, select the **Loop** option in the "Create Record" step. The Loop option has three modes: Bulk, Sequential, or Parallel. Provide the list of JSON inputs containing the sourcedata as the input to the loop and refer to each element as `{{ vars.item }}` in the step. For example, if you can provide the following JSON as input to the Loop option in the Create Record step to create alerts in FortiSOAR: `[{"name": "Name 1", "source": "FortiSIEM"}, {"name": "Name 2", "source": "FortiSIEM"}]`
You can ensure that the two alerts created in FortiSOAR have the corresponding names by using `{{ vars.item.name }}` against the `Name` field in the step.
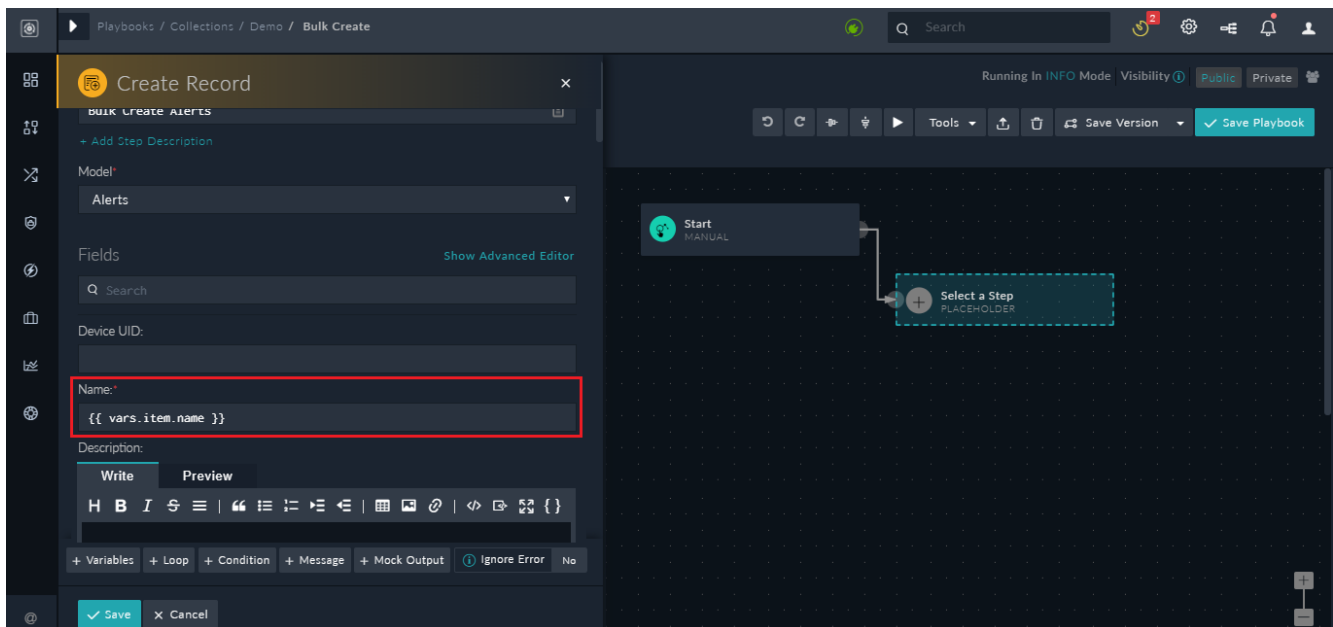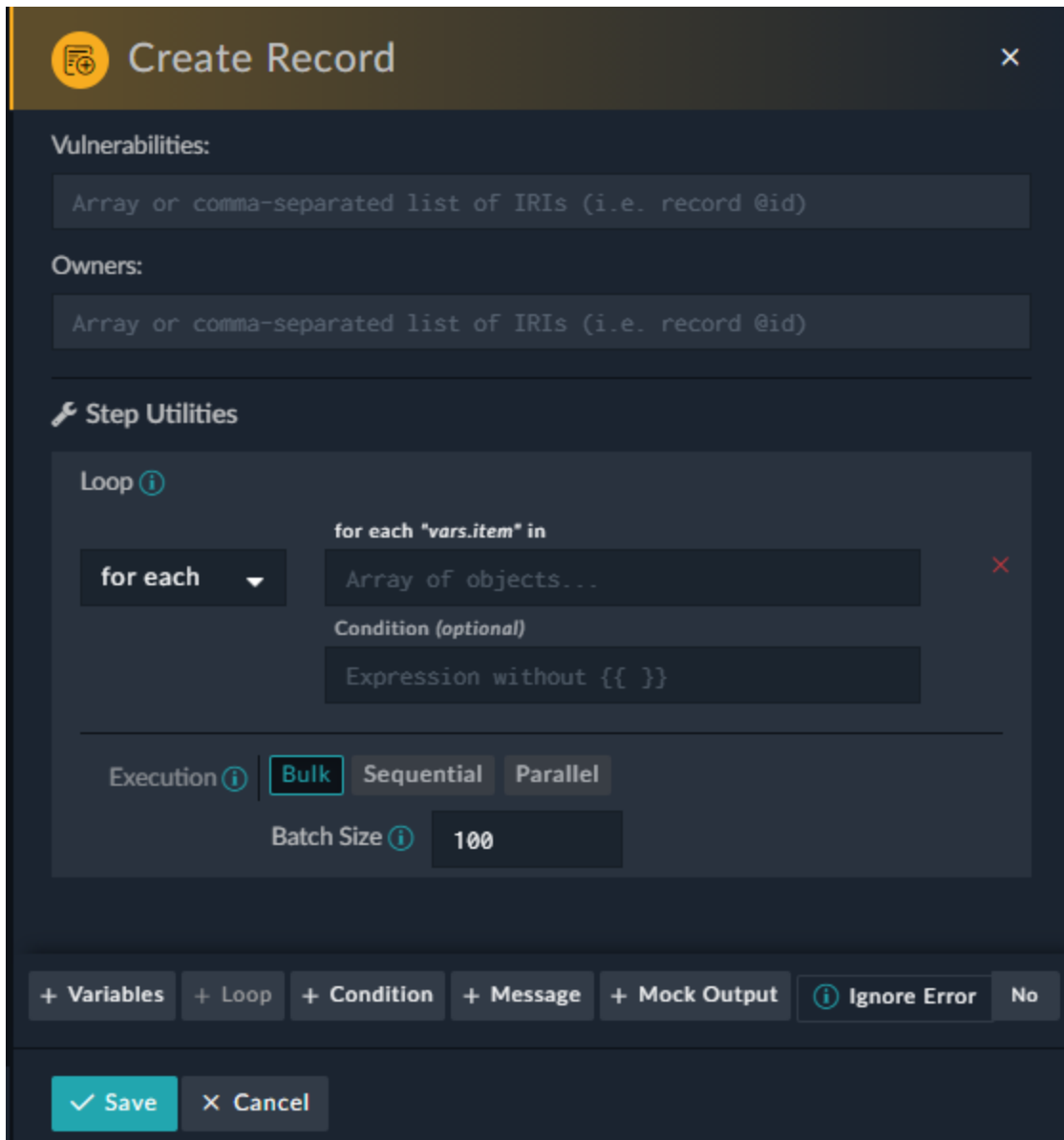
The **Bulk** mode creates all records in a single API request and is the most optimal and recommended method of creating or upserting records in bulk. This is also the default mode when you add a new Create Record step in a loop. The **Sequential** modes sends the API records separately for individual records, and one after another. So, the playbook step can abort at the first failure, without proceeding to create further records. The **Parallel** modes sends separate API requests for each record creation but using multiple threads to do so.

Batching large datasets when using the 'Bulk' option

A single batch can handle 100 to 200 records depending on the record size. If you are inserting larger number of records that causes the API call to time out, then you can insert records in batches.

From version 7.0.0 onwards, the 'Bulk' option has been enhanced to support batching of large number of records, by default, in the Create/Update record steps. To support this, the '**Batch Size**' option for the Bulk execution type has been added making it easy to bulk insert, upsert, or update large number of records. By default, the batch size is set to 100 records. You can increase or decrease this batch size depending on the record size. The following image shows a sample 'Create Record' step that is inserting a batch of 100 records:

Update Record

Use the **Update Record** step to update a record in a module within FortiSOAR.

In the Playbook Designer, click the **Update Record** step and add the step name in the **Step Name** field, add the field to be updated in the `resource` field, add the module name and UUID of the record to be updated in the `collection` field (for example, you want to update the `Alerts` module, you will enter `api/3/alerts/{{uuid}}`), and then click **Save**.

The UI of the **Update Record** step displays an `Update Record` form that contains fields depending on the module you select in the **Model** drop-down menu, like the Create Record Step.

You must add the UUID or IRI of the Record you want to update in the **Record ID** field. In the **Record ID** field add either the IRI of the record that you want to update or add the jinja value of the record.

You can add details and field values to the "Update Record" step similar to the "Create Record" step.

If in the Update Record step, you are specifying any `Date/Time` field in the jinja format, then that date/time field must in the `epoch` format similar to the "Create Record" step. You can use the methods described in the "Create Record" step to convert the input date/time field to the epoch time. However, there is a difference between the "Create Record" step and the "Update Record" step, if you choose to enter the data directly in the JSON format by clicking the **Add Custom Expression** button, which displays `Dynamic Values`. Dynamic Values appears empty in the case of Update Record (unlike the Create Record step, which displays fields according to the module you have selected) since you require to add only those fields in the JSON format that you want to update and do not require to see all the fields.

Once you add the record ID, you can update any of the fields of that record in the `Update Record` form directly and click **Save**. Once you click save, the data in the record that you specify by the record ID gets updated based on the changes you have made.

Find Records

Use the **Find Records** step to find a record in a module within FortiSOAR, using a query or search criteria.

In the Playbook Designer, click the **Find Records** step, and add the step name in the **Step Name** field, and then select the module in which you want to search for the record in the `Module` field.

The Find Records step by default fetches only 30 records, if you want to change the number of records to be fetched, then enter the number of records to be fetched in the **Record Limit** field. For example, in the following image above we have entered `100` in the **Record Limit** field, which means that up to 100 records will be fetched.

> From version 7.0.0 onward, the number of records that can be fetched using the 'Find Record' step has been limited to 5000. To override this default number, which is *Not Recommended*, you need to follow the steps mentioned in the Increasing the number of records that can be fetched using the 'Find Record' step topic.

To include records that are correlated with the records that are being fetched using the Find Records step. If you want to include correlated record, then select **True** from the **Include Correlated Records** field. By default, the **Include Correlated Records** is set to **False** for performance efficiencies.

You can select the **Limit Output** checkbox, to limit and refine your search results to only those fields that you require allowing for better usability and performance. For example, if you want to limit the output to display *only* the "name of the record" and "related indicators", then you should select the **Limit Output** checkbox and in the selection box that follows the Limit Output checkbox, select **name**, and **indicators**.
Note: If **Include Correlated Records** is set to **False**, then you can select only fields of the selected module. Therefore, to include related indicators, you must ensure that you set **Include Correlated Records** to **True**, which would then display all the correlated records.

From the **Module** drop-down list, select a module in which you want to search for records. Once you select the module, the **Nested filters** component appears in the `Build Search Query` section using which you can build the search query to find records and the click **Save**.
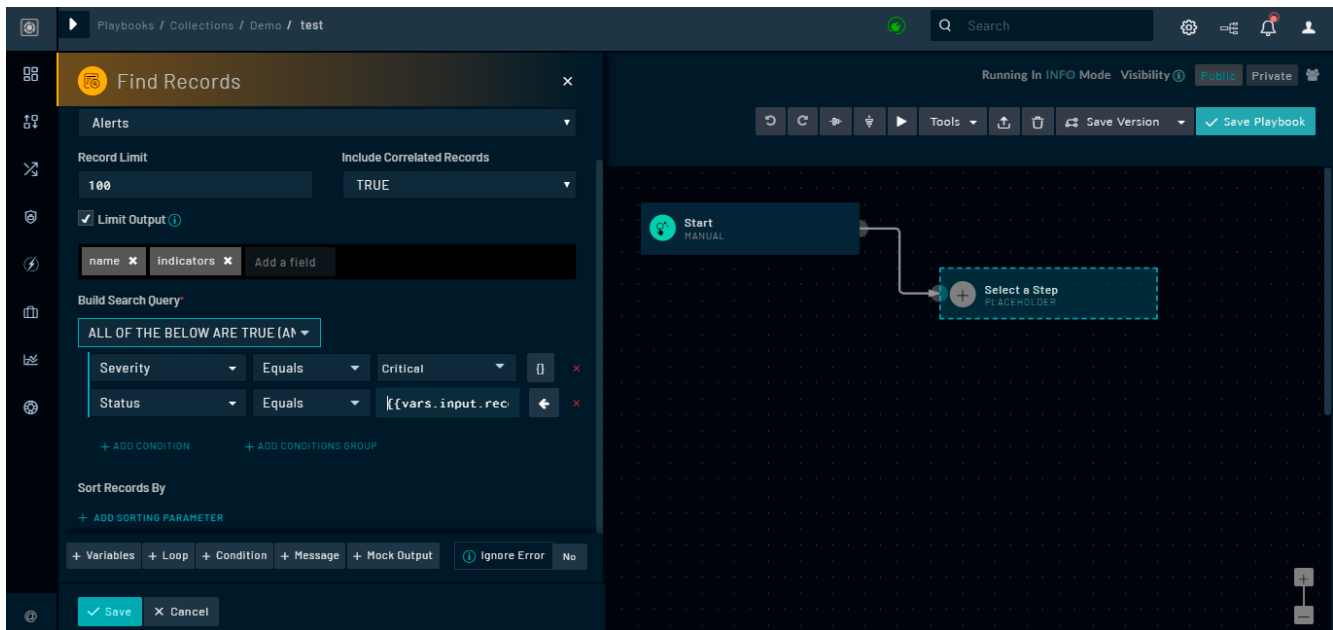
You can use **Nested filters** to filter records using a complex set of conditions. Nested filters group conditions at varying levels and use **AND** and **OR** logical operators so that you can filter down to the exact records you require.

> If you assign a "Custom" filter to a datetime field, such as Assigned Date, then the date considered will be in the "UTC" time and not your system time.

For more information on nested filters, see the *Nested Filters* topic in *Dashboards, Templates, and Widgets* in the "User Guide."

> You cannot search or filter encrypted fields.

You can also write Jinja to build your search query in the **Nested filters** component in the `Build Search Query` section. You can either write you own Jinja or use the Dynamic Values dialog to add jinja to the field. See the Dynamic Values chapter for more information. You can also toggle between the Jinja and the original field type, for example in the image above; the **Severity** field displays the field as a drop-down list (which is the original field type). Click the [] icon to enter Jinja for this field. Similarly, the **Status** field displays the Jinja that has been entered in the field. Click the ◄ icon to toggle back to the original field type, which is a drop-down list.

You can also search records using a UUID. To search using UUID, in the **Nested filters** component in the `Build Search Query` section, select **UUID** from the Select a field drop-down list, select the operator such as **Equals** from the Select Operator drop-down list, then click the filter field to display the `Dynamic Values` dialog. Click the **Expressions** tab and then click the **IRI to UUID** expression:
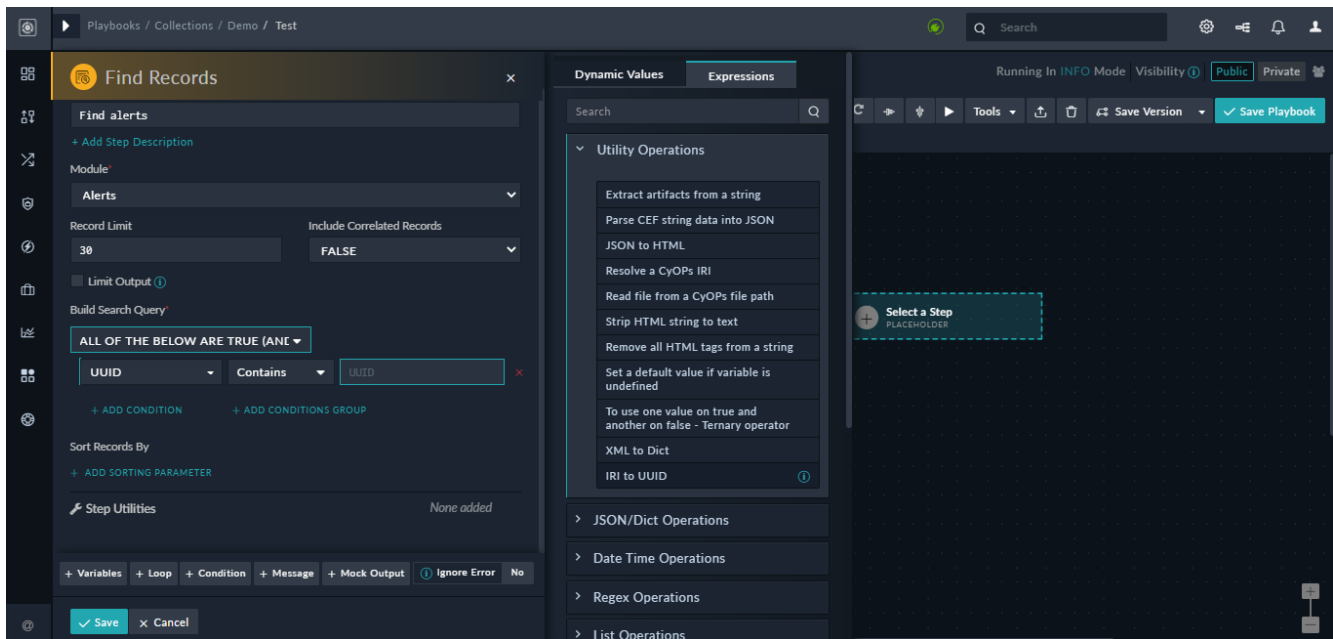
In the `Utility Operations: IRI to UUID` popup, enter a valid FortiSOAR IRI and click **Add**:
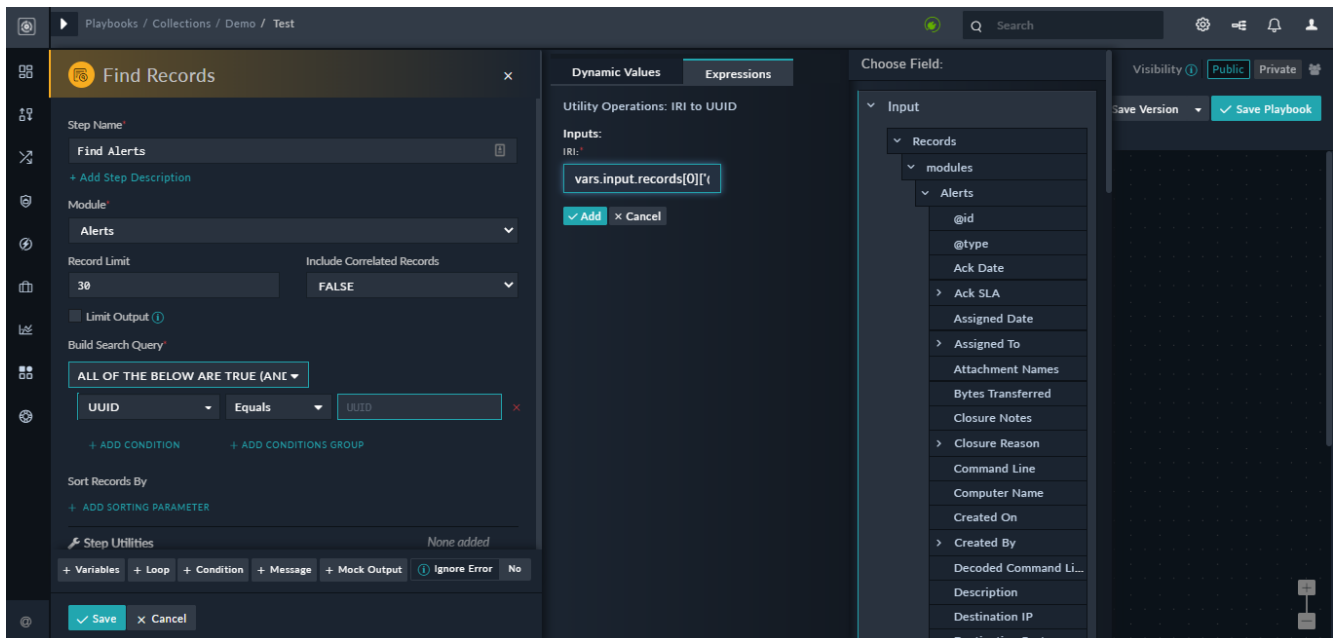
You can either add the IRI value directly or again use Dynamic Values to enter a jinja expression for the IRI. For more information, see the Dynamic Values chapter. The `Utility Operations: IRI to UUID` converts a valid IRI to a UUID using which you can search for records.

You can also sort the fetched records easily by clicking the **Add Sorting Parameter** link and choose the field based on which you want to sort the records in the `Sort Records by` section. You can also specify whether you want to sort the records in the **Ascending** or **Descending** order. For more information on sorting records, see the *Default Sort* topic in *Dashboards, Templates, and Widgets* in the "User Guide."

Increasing the number of records that can be fetched using the 'Find Record' step

From version 7.0.0 onward, the number of records that can be fetched using the 'Find Record' step has been limited to 5000. It is not recommended to change the value to a higher number and instead, you should use pagination by making an API call and navigate to the next page in a loop. However, if you yet want to override this default and increase the number of records to be fetched, do the following:

1. Edit the `/opt/cyops-api/config/parameters_prod.yaml` file to add the following code:
```
api_platform:
    collection:
      pagination:
        maximum_items_per_page: 5000
```
Change the value of the `maximum_items_per_page` parameter from '5000' to the desired number.
Important: The above code must be added at the same level as '`parameters`' in the `parameters_prod.yaml` file.

2. Restart php-fpm and run cache clear:
```
# systemctl restart php-fpm
sudo -u nginx php /opt/cyops-api/bin/console cache:clear
```

Set Variable

Use the **Set Variable** step to record a specific variable or variables for future use. Enter the variable name in alphanumeric characters and then define the value. The value may be a dynamic value itself. The scope of the variable created using `Set Variable` is *local*.

To create a variable, in the Playbook Designer, click the **Set Variable** step and add the `Name` and `Value` for the variable and then click **Save**. You can define multiple set variables in a playbook.

> ⚠️ Do not use the following reserved words as a variable name in the Set Variable step: `for_each`, `do_until`, `ignore_errors`, `condition`, `message`, and `mock_result`.

Once defined, the variable can be referenced in any remaining steps or in any child playbook, regardless of how many levels deep, the child playbooks are called.

The format for calling a variable is `{{vars.%name%}}`.

> 💡 You can declare variables directly in the step, using the Variables option. See Variables for more information.

Evaluate

Decision

The **Decision** step serves as conditional validation within the playbook. You can specify "if this, then that" criteria that directs the playbook to execute specific steps based on the results of a specific condition. Many organizational processes differ depending on particular criteria, and to accomplish this; you can use the Decision step.

Use the Decision step to allow the playbook to specify, "If criteria = x, then do this next step." However, you can configure the Decision step with a variety of operators (equals, does not equal, <, >, etc.) and you can even chain logical conditions with AND/OR logic, allowing the organization's playbooks to define granular specifications for executing a specific sequence of steps.

To add a decision step to a playbook, click the **Decision** step. Initially, the Playbook Designer displays only the **Step Name** field, with no conditions. Type the step name and click **Save**. You can either create a Decision step with just the Step Name specified for now or create the possible conditions first then create the Decision step and then identify the condition once the Decision step and the potential outcome steps are connected. You can also define the entire step setting, or workflow, for the decision step, even if the connecting step is unavailable, allowing you to write the complete logic of the decision and then plug in the steps later.

The decision step functions in such a way that it evaluates multiple (alternative) conditions until any of them is fulfilled. This means that when the **Decision** step finds one condition that is fulfilled, then it skips the other conditions. If none of the conditions are fulfilled, then the default condition or route is defined.

You can define a default route that the playbook should take if none of the defined conditions are fulfilled. You can also add a description to describe the various routes that can be taken by the playbook.

Example

A playbook execution route is based on the severity of the alert that gets created in the system at the hands of a third-party integration. If the alert that is created is not assigned any severity or a severity that does not match the severity that is defined in any of the conditions, i.e., Low or Minimal in our example, then it follows a default assignment, which is that the alert record assignment is updated to a Tier1 Analyst. This would be **Step A Alert Assigned to Tier1 Analyst**. If the alert is created with *Critical* severity, then that alert gets assigned to an Administrator (**CS Admin**). This would be **Step B Alert Assigned to CS Admin**. If the alert is created with *High* or *Medium* severity, then that alert gets assigned to a Tier2 Analyst. This would be **Step C Alert Assigned to Tier2 Analyst**.

The steps to create a playbook based on the above example are as follows:

1. Create a playbook, for example, `Alert Assignment` Playbook.
    FortiSOAR displays the Playbook Designer. The procedure for creating playbooks is mentioned in the *Playbooks Overview* section.
2. Add a On Create trigger, by clicking **On Create Trigger** in the `Playbook Designer`, type the name of the step in the **Step Name** field, for example, `Alert Creation` and then from the **Resource** drop-down list select the module on whose creation you want to trigger the playbook, for our example select `Alerts` and then click **Save**.
3. Drag-and-drop a connector point to connect to another playbook step. FortiSOAR adds a placeholder step on the playbook designer page and opens the `Steps` tab which displays all the available playbook steps. Click the **Decision** step and type the step name as `Assignment Based on Severity` and click **Save**.
4. Drag-and-drop connector points from the `Assignment Based on Severity` decision step and create the routes that the user can follow, i.e., create Step A, B, and C
5. Create Step A, where the alert is created with no severity, as follows:
    a. Click **Update Record**.
    b. Type the name of the step in the **Step Name** field, for example, `Step A Alert Assigned to Tier1 Analyst`.
    c. From the **Model** drop-down list, select **Alerts**.
    d. In the **Record IRI** field, use the Dynamic Values dialog and select the current record, `Input > Records > module > alerts > @id`.
    e. From the **Assigned To** drop-down list, select **Tier1 Analyst**, and click **Save**.
        Note: The DateTime field in a playbook step, for example, in a condition step, does not have the "Is Null" option in the Select Operator drop-down list.

6. Create Step B, where the alert is created with Severity Equal to Critical, as follows:
    a. Click **Update Record**.
    b. Type the name of the step in the **Step Name** field, for example, `Step B Alert Assigned to CS Admin`.
    c. From the **Model** drop-down list, select **Alerts**.
    d. In the **Record IRI** field, use the Dynamic Values dialog and select the current record, `Input > Records > module > alerts > @id`.
    e. From the **Assigned To** drop-down list, select **CS Admin**, and click **Save**.
7. Create Step C, where the alert is created with Severity Equal to High, as follows:
    a. Click **Update Record**.
    b. Type the name of the step in the **Step Name** field, for example, `Step C Alert Assigned to Tier2 Analyst`.
    c. From the **Model** drop-down list, select **Alerts**.
    d. In the **Record IRI** field, use the Dynamic Values dialog and select the current record, `Input > Records > module > alerts > @id`.
    e. From the **Assigned To** drop-down list, select **Tier2 Analyst**, and click **Save**.
8. Add conditions to the **Decision** step as follows:
    a. **For the Default Step**:
        i. Click **Add Default Condition**.
        ii. From the **Select A Step to Execute**, select **Step A Alert Assigned to Tier1 Analyst**.
           You can optionally also add a tooltip, that describes the route this step or condition will take.
    b. **For the alternative steps**:
        i. Click **Add Condition**.
        ii. In the `Condition 2` section, use the *Condition Builder* to build the Severity Equals Critical condition as follows: From the **Select a field** drop-down list, select **Severity**, from the **Operator** drop-down list, select **Equals**, and from the **Select** drop-down list, select **Critical**.
           If you want to use jinja to add advanced expressions and create complex conditions, you can click the **Show Advanced** link and add jinja in the condition text box.
        iii. From the **Select A Step to Execute**, select **Step B Alert Assigned to CS Admin**.
           You can optionally also add a tooltip, that describes the route this step or condition will take.
        iv. Click **Add Condition**.
        v. In the `Condition 3` section, use the *Condition Builder* to build the Severity Equals High or Medium condition as follows: From the **Select a field** drop-down list, select **Severity**, from the **Operator** drop-down list, select **Equals**, and from the **Select** drop-down list, select **High**, and from the **Select Logical** drop-down list select **Or**, and then select **Severity**, from the **Operator** drop-down list, select **Equals**, and from the **Select** drop-down list, select **Medium**.

vi. From the **Select A Step to Execute**, select **Step C Alert Assigned to Tier2 Analyst**.
You can optionally also add a tooltip, that describes the route this step or condition will take.
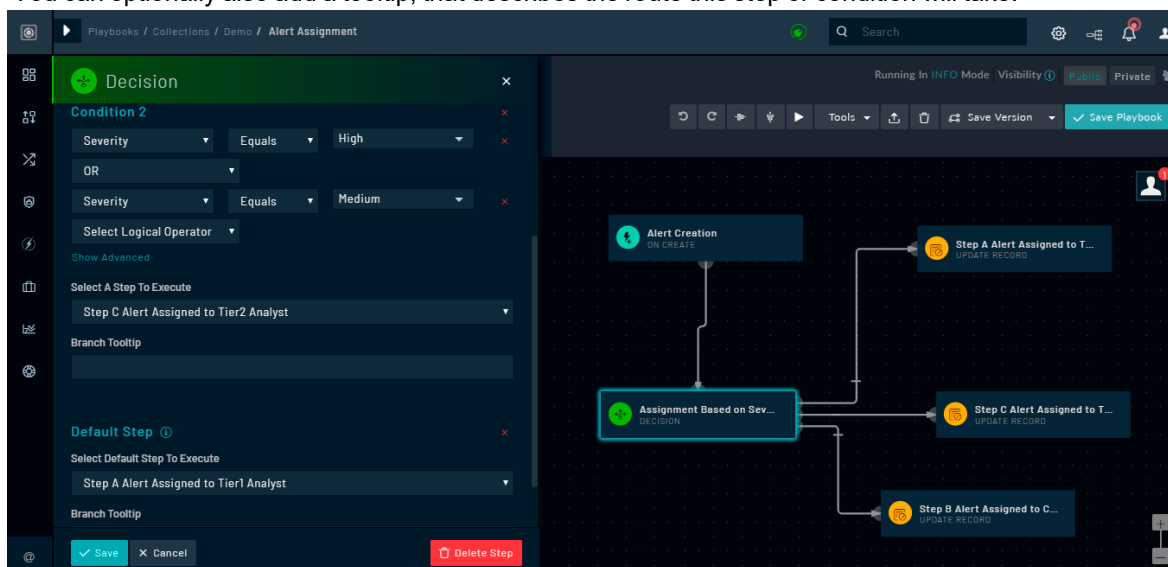
In case of the No Trigger step for the **Condition Builder** you must add advanced jinja expressions in the **Condition** field. In the case of the *Manual Trigger*, if you have selected multiple modules, then for the **Condition Builder** you must add advanced jinja expressions in the **Condition** field.

Wait

Use the **Wait** step to specify the time that the playbook should wait after a specific step and before continuing with the remaining steps in the playbook. This is useful for defining specific periods to wait for an action to occur in an external system or to allow for SLAs to elapse before continuing on the course of the playbook.

To add a delay to a playbook, click the **Wait** step and in the **Step Name** field, type the name of the step, and optionally add a description of the step in the **Description** field. In the `Playbook will resume after` section, enter the values in the **Weeks**, **Days** or **H:M:S** (Hours, Minutes, and Seconds) fields, which specifies the time till when the playbook will wait before executing the remaining steps in the playbook and then click **Save**.

Once you have saved the step and the **Wait** step appears on the Playbook Designer canvas, place the **Wait** step in between steps that require a delay.

If a child playbook contains a "Wait" step, then it runs synchronously with the parent playbook, i.e., the parent playbook will wait for the child step to complete and only then resume its workflow. Earlier, if a child playbook contained a wait step, it would run asynchronously from the parent playbook, i.e., the parent playbook would continue its workflow independent of the child playbook and without waiting for the child playbook to complete its workflow.

Approval

Use the **Approval** step to halt the execution of Playbook steps until an approval is received from the person or team that you have specified as an approver. Only once the approval is received will the Playbook move ahead with the workflow as per the specified sequence. Until the approval is not received, the **Execution Playbook Logs** will display the Playbook status as `awaiting`. Once you complete adding an approval step, which includes adding the approver, the

approver gets a notification for approval and the approver either accepts or rejects the approval request. Once an approval request is complete, the original playbook that contains the approval step resumes the execution of the remaining playbook steps. You can select only a single team or user as an approver.

The approval step includes **Email** as a mode of approval, apart from the default, which is system notification.

Permissions Required

- To view Approval notifications and provide approvals, you must be assigned a role that has a minimum of `Read` and `Update` permissions on the `Approvals` module and `Create` and `Read` permissions on `Playbooks` module.
- To create a playbook and add an approval step or any other step, you must be assigned a role that has a minimum of `Create`, `Read` and `Update` permissions on the `Playbooks` module, and a minimum of `Read` permissions on the `People` and `Security` modules.
- To add an approval step or any other step to an already existing playbook, you must be assigned a role that has a minimum of `Read` and `Update` permissions on the `Playbooks` module, and a minimum of `Read` permissions on the `People` and `Security` modules.

Examples of usage of an approval step:

- A case for when you can use an approval step could be when you have sent a URL to a third-party URL authenticator to identify whether the URL is malicious or not. If you get a report from the third-party URL authenticator that the URL is malicious, then you want to block that URL. However, before you block that URL, you require approval from your SOC manager and therefore here you would use an approval step.
- Another example would be when you want an Incident to be deleted from the system. However, before the deletion, you require approval from an Incident Lead and therefore here you would use an approval step.

Adding an approval step for System notification

1. Open FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click on an existing playbook collection.
   This opens the `Playbook` page, click on the playbook in which you want to add an approval step.
   This opens the playbook in the `Playbook Designer`.
3. Click the **Approval** step in the `Evaluate` section.
4. For the Approval step, in the **Step Name** field, add the name of the step.
5. From the **Choose Approver** drop-down list, choose **Team** or **User**, which displays a **Select** link. Clicking the **Select** link displays a `Team` or `User` pop-up based on the approver type you have chosen.
   The Team or User pop-up lists all the existing teams or users. Select the team or user who can provide approval. If

you select **Team**, then any member of that team can provide the approval.
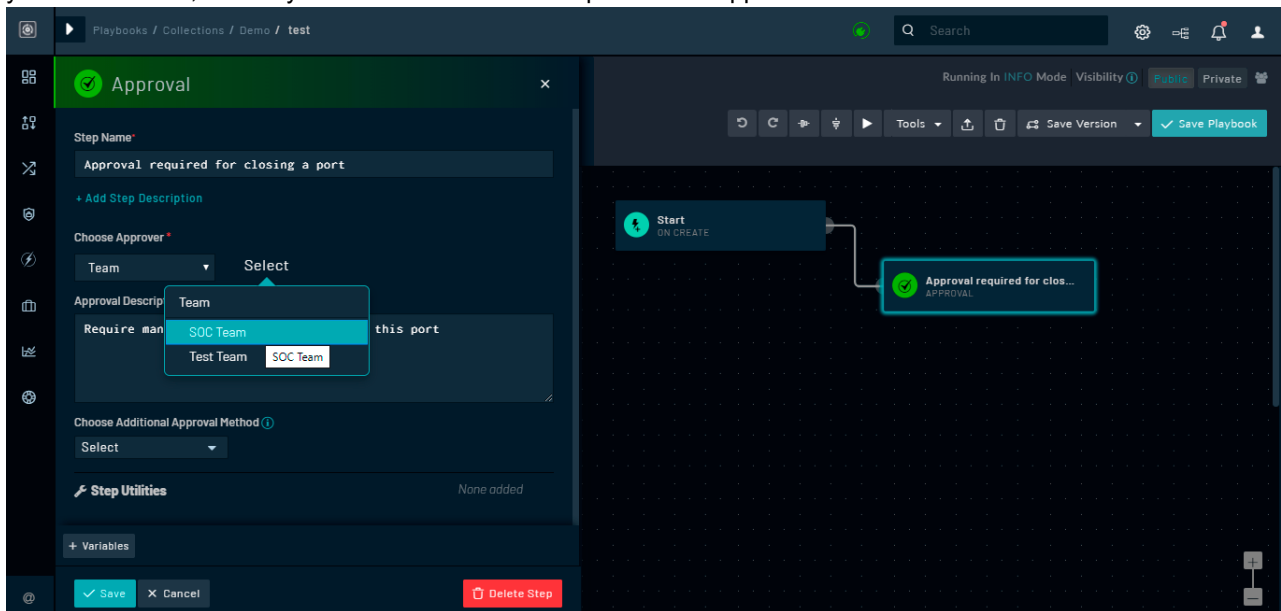
6. In the **Approval Description** field, add the description of the approval request, can include the reason for the approval request.
    The approvers can view this description in the approval notification.

7. (Optional) To add email as a mode of approval, apart from the default, which is system notification, from the **Choose Additional Approval Method** drop-down list, choose **Email**.
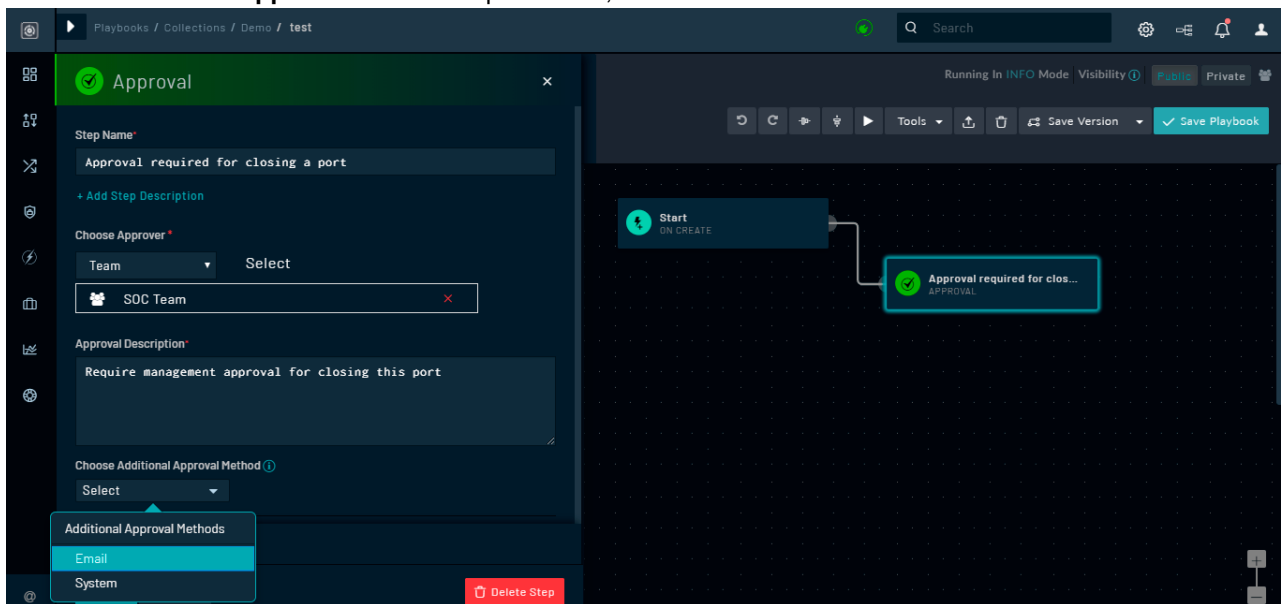
Note: You can also add your custom approval modes such as SMS or other third-party integrations.

8. Click **Save** to save the approval step.

Playbook status with respect to Approval

Click the **Executed Playbook Logs** icon in the upper-right corner of FortiSOAR to view the logs and results of your executed playbook. Clicking the **Executed Playbook Logs** icon displays the Executed Playbook Logs dialog as shown in the following image:

- Until the approval is not received, the **Execution Playbook Logs** will display the Playbook status as `awaiting`.
- If the approval is rejected or granted, the **Execution Playbook Logs** will display the Playbook status as `finished` and the playbook continues to execute remaining steps, as defined for approval rejection or acceptance, and if the playbook successfully completes executing all the steps, the **Execution Playbook Logs** will display the Playbook status as `finished`.

For information on Execution History, see the Debugging and Optimizing Playbooks chapter.

Approval notification using the System mode

Once you complete adding an approval step which includes adding the approvers, the approvers get a notification for approval. Users who have the appropriate permissions for approval receive the notification. The **Notification** icon is present on the top-right corner in FortiSOAR, and when a user has a notification, the icon blinks with a red light. Click the icon to open the list of your notifications. In the case of approvals, the notifications, appear as `Approvals`. Clicking **Approvals** displays the Playbook Name and the Description for the approval request.

The approver clicks **Approve** or **Reject** to approve or reject the request. The approver can also optionally add comments in the **Comments** field that explain the reason for the approval or rejection of the request.

When a user logs into FortiSOAR and uses the system method to approve a request, FortiSOAR displays 'Unauthorized access', though the original playbook resumes the execution of the remaining playbook steps and moves to the 'finished' state (if there are no further errors in the playbook). This is because the 'Approval' module inserts an approval record using a playbook the ownership of that record always remains with the SOC team (admin team) irrespective of the team or user who triggered the playbook. To solve this issue, you must add the team(s) who will provide approval to be part of 'Appliance.' For more information about Appliances, see the 'Security' chapter in the "Administration" Guide.

Once an approver completes an approval request, the notification dialog displays the approval request as approved using a green check symbol and the notification is removed from the notification window.

Approval notification using the Email mode

If you have chosen **Email** in addition to system as a mode of approval, an email will be sent to the email ID that has been configured for the users in their profile. See *Security Management* in the "Administration" guide for more information on configuring user profiles.

If you have selected a team to provide approval in the approval step, then the email notification is sent to all the team members, who have appropriate permissions, and any of the team members can provide approval.

The approval email notification contains a link to an Approval Request dialog, which contains the name of the playbook from which the request has been sent and also the description of the approval required. Once an approver clicks on the link in the email the `Approval Request` dialog is displayed, and the approver can click **Approve** or **Reject** to approve or reject the approval request. The approver can add comments in the **Comments** field that explain the reason for the approval or rejection of the request.

Users can choose to approve the request using the system mode as well since apart from the email notification; a system notification is also sent for the request. If a user uses the system method to approve a request, then FortiSOAR displays 'Unauthorized access', though the original playbook resumes the execution of the remaining playbook steps and moves to the 'finished' state. This is because when the approval record is inserted using a playbook, then the ownership of that record always remains with the SOC team (admin team) irrespective of the team or user who triggered the playbook. To solve this issue, you must add the team(s) who will provide approval to be part of 'Appliance.' For more information about Appliances, see the 'Security' chapter in the "Administration" Guide.

Once an approver completes an approval request, using any mode of approval, the notification dialog displays the approval request as approved using a green check symbol and the notification is removed from the notification window.

Viewing details of an approval record

Once you trigger a playbook a record for the same is created in the Approvals module, and you can view and edit the details of the approval in this record as shown in the following image:

The `Approvals` module is not included as part of the default modules. Therefore, you must add the Approval module using the **Navigation Editor** if you want the Approvals module to appear in the FortiSOAR left navigation. For information on how to add modules to the FortiSOAR left navigation, see the *Navigation Editor* topic in the "Administration Guide."

You can edit details of approval add or edit the description of the approval or update the approval or rejection message. You can also reassign the task of approval to another user in cases such as the user to whom the approval was originally assigned is unavailable.

> When you reassign the approval to another user that user will not get the notification of that assignment unless you have chosen **Email** as the additional method of approval while configuring the **Approval** step. If you have only configured the **System** method of approval, then the reassigned user will not get an approval request notification.

Viewing details of the approval playbooks

You can view the details of the approval by clicking the *Execution History* tab to view the logs and results of your executed playbook. For more information on Playbook Execution History, see the Debugging and Optimizing Playbooks chapter.

Using the output of the Approval step in other playbook steps

To use the output of the approval step in other playbook steps or to display the result of the approval step in the **Step Results** option in Dynamic Values, you must add the following jinja to the step that requires to use the output of the `Approval` step:

- To get result of the approval, i.e. true or False: `{{vars.steps.<nameOfTheApprovalStep>.approved}}`
- To get the comment or message associated with the approval: `{{vars.steps.<nameOfTheApprovalStep>.message}}`
- To get the user who is the approver: `{{vars.steps.<nameOfTheApprovalStep>.user}}`

Manual Task

Use the **Manual Task** step to pause the execution of the playbook till you complete a manual task such as a manual shutdown of a server, or starting or stopping a firewall, that is part of an automated workflow.

Once you click the **Manual Task** step, a form containing the fields from the `Task` module is displayed. Enter content for the fields in the `Task` module, such as the name of the task, person to whom the task is assigned, the status of the task, and the date by when the task is to be completed. Once you click **Save**, this record is added in the `Task` module, and a FortiSOAR system-playbook begins to run in the background, which keeps checking the status of this task.

Once a user changes the **Status** of the added manual task in the `Task` module, to either **Skipped** or **Completed**, then the system-playbook gets notified about the status change and in turn the system playbook resumes the execution of the original playbook that had requested the manual task.

Note: You can change the condition for when the manual task should resume, for example, you can specify that the manual task should resume only when the user changes the **Status** of the manual task to **Completed**. You must update the `System` playbooks if you want to configure the manual task conditions. You can view system playbooks by clicking the **Settings** icon, then clicking the **System Configuration** option, and then clicking the **System Fixtures** tab. On the `System Fixtures` page, in the `Playbooks` section, click **Approval/Manual Task Playbooks**.
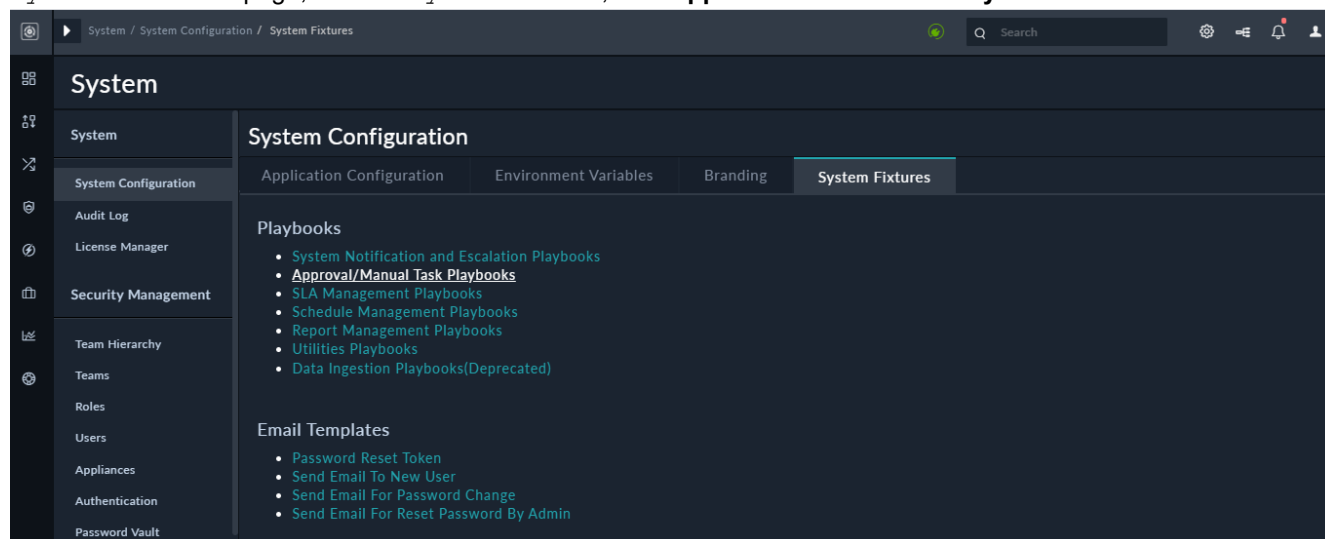
Using the output of the Manual Task step in other playbook steps

To use the output of the manual task step in other playbook steps or to display the result of the manual task step in the **Step Results** option in Dynamic Values, you must add the following jinja to the step that requires to use the output of the `Manual Task` step:

- To get the ID of the manual task: `{{vars.steps.<nameOfManualTaskStep>['task_data']['@id']}}`
- To get status of the manual task: `{{vars.steps.<nameOfManualTaskStep>.status}}`

Manual Input

Use the Manual Input step to display a customized pop-up either for user prompt or decision anywhere in the flow of the playbook. Based on the input or decision that the user takes, the playbook will choose one of the paths, from the paths that you have defined in the playbook and continue to execute the playbook as per the specified automated workflow.

The manual input step can be used with all types of playbook triggers, including Custom API Endpoint trigger and Referenced trigger.

In case of an Custom API Endpoint trigger, a Referenced trigger, and a Manual trigger that has been created with the **Run Without Selecting Any Record** option selected, you must specify the module and the record IRI on which the action has to be taken. The module specified will also be used to populate the "People" lookup and assign ownership to specific users or teams.

The Manual Input step provides you with the ability to configure two types of input prompts: Decision-based prompts and Input-based prompts.

An example of an input-based prompt would be the enrichment of indicators associated with an alert record in FortiSOAR that has been generated from a SIEM. Enrichment of indicators would be done using threat intelligence tools, for example VirusTotal. The results from VirusTotal state that there are 3 indicators, 2 of which are marked as suspicious based on their score received from VirusTotal and 1 marked as marked as malicious based on their score received from VirusTotal. The Manual Input step would list these 3 indicators and prompt SOC analysts for an evaluation of indicators and select the ones that they think should be marked as malicious. Based on the analyst's evaluation further action will be taken on the alert record and the associated indicators. From version 7.0.0 onwards, you can add visibility conditions to the fields displayed in the user input form, i.e., fields in the user form would be visible based on the specified conditions. For an example of using the visibility conditions in a user prompt, see the User Input Prompt - Visibility Conditions section.

From version 7.0.2 onwards, **Dynamic list** is supported as a **'Custom'** input type in both the Manual Trigger step and the Manual Input step. A 'Dynamic list' is a list with dynamic values that is set using a playbook, i.e., the options of the list are defined using JSON or comma-separated values, set using Jinja and gets displayed as a list. For an example of how to use dynamic lists in user prompt, see the User Input Prompt - Dynamic Lists section.

An example of a decision-based prompt would be similar to the above except that there would be generally be a question in the prompt based on which the SOC analyst would require to make a decision. For example, Is the following Indicator Malicious? The analyst then just has to choose either Yes - Block the Indicator or No - Do not block the indicator. Based on the SOC analyst decision further action will be taken on the alert record and the associated indicators. You can also retrieve the reputation of the indicator from various threat intelligence tools such as VirusTotal using the Connector step and display this information to the analysts to enable them to take a more informed decision.

Building a decision-based input prompt

Perform the following steps to create a playbook with a Manual Input playbook based on prompting SOC analysts for an evaluation of indicators that are associated with an alert generated in FortiSOAR and confirm whether they are malicious or not. Based on the analysts' evaluation further action is taken on the alert record.

1. Open FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click on an existing playbook collection.
   This opens the `Playbook` page, click on the playbook in which you want to add the **Manual Input** step, or add a new playbook, which opens the playbook in the `Playbook Designer`. For our example, create a new playbook named `Test - Manual Input` and ensure that the **Active** checkbox is clicked, then click **Create**.
   This opens the Test - Manual Input playbook in the Playbook Designer.
3. In the Trigger step, select **Manual Trigger** and define the following parameters:
   a. In the **Step Name** field, enter the name of the playbook. For example, `Test Manual Input`.
   b. In the **Trigger Button Label** field, type the playbook name as `Indicator Evaluation`.
   c. Ensure that the **Run once for all selected records** option is selected.
   d. In the **Choose record modules on which the playbook would be available on** field select the **Alerts** module.
   e. Click **Save**.

4. To get the indicators associated with the record, you can add a **Find Records** step with the **Indicator** module selected and in the `Build Search Query` section, select Alerts in the Related module and then using Dynamic Values add the condition as `ID Equals {{vars.input.records[0].id}}` and click **Save** to save the step.

5. Add the steps that you want to add as the response actions to the evaluate the inputs provided by the analysts. For our example, configure the **Block Indicator** and **Send Email Notification** steps as per your requirements.

6. From the `Evaluate` section, choose **Manual Input** and define the following parameters:

   a. In the **Step Name** field, enter the name of the playbook. For example, `Manual Input Test`.

   b. In the `Input Prompt Configuration` section, select **Decision-Based**.
   From version 7.0.0 onwards, you can choose to request decisions or other inputs from non-FortiSOAR users. To allow non-FortiSOAR users to provide decisions and inputs, click the **Create a link for users outside FortiSOAR** checkbox.
   You can define the following parameters for the decision-based prompt:

   i. In the **Title** field, enter the title for the prompt. For example, `Indicator_Malicious or not`.

   ii. In the **Description** field, enter the description for the input prompt. For example, `Should the indicator be blocked?` and add then add the jinja to retrieve the indicators associated with the alert in the format: `{{vars.steps.<nameOfFindRecodsStep[0].value}}`. For example, `{{vars.steps.Find_Associated_Indicators[0].value}}`
   Use **Dynamic Values** to add jinja in playbooks. For more information, see the Dynamic Values chapter.

   iii. Click **Save** to save the step.
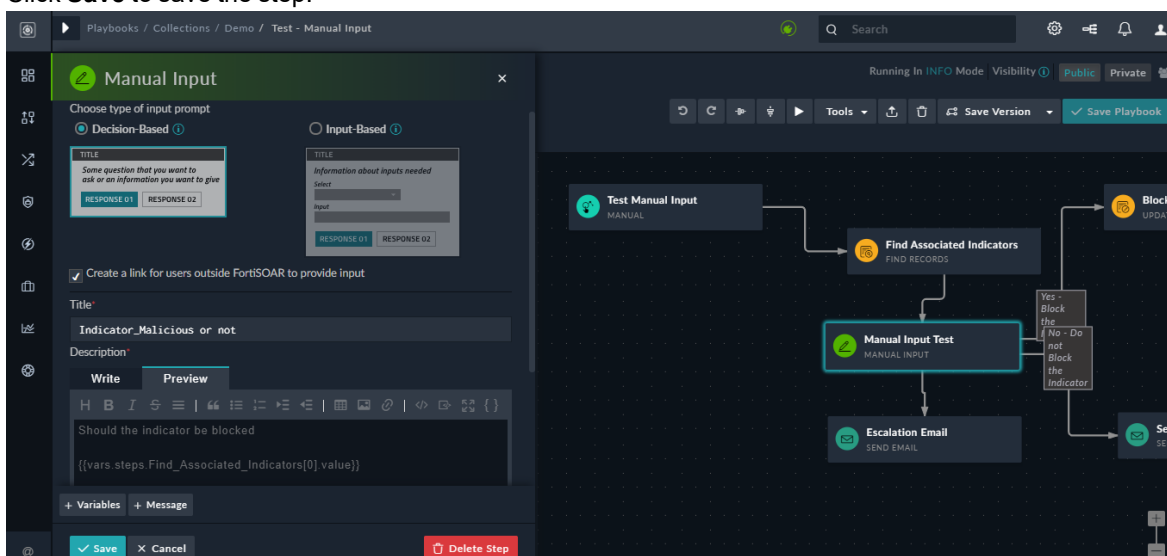
   c. In the `Response Mapping` section, you add the custom response options that the user can choose from when presented with the decision. You should map each custom response option to a corresponding playbook step. You can add the custom response for the decision first so that you can define the complete workflow and then create the corresponding playbook steps.
   For our example, in the `Response to Step Mapping` section, click the **Add More** link and in the **Response** field, type `Yes - Block the Indicator` and corresponding to this response, in the **Choose Step** field, select the **Block Indicator** step to block the indicator . Then click **Add More** and type `No - Do not Block the Indicator` and corresponding to this response, in the **Choose Step** field, select the **Send Email Notification** step to send a notification to the SOC team (admin team) so that they can be informed that this indicator is not blocked and they can take further steps if required.
   You can also select the response that you want to consider as a primary response by selecting the **Primary** checkbox. Selecting a response as primary will add a distinct visual style to that option button, making it more prominent when compared to the other buttons. In our example we have marked the **No - Do not Block the**

Indicator option as the primary response.

d. If you have not selected the **Create a link for users outside FortiSOAR** checkbox, then you will see `Ownership` section, else you see the `Email Recipients` section.

- In the `Ownership` section, you can specify if you want a specific user or team to be responsible for responding to the user prompt and provide the input or decision.
  If you select **No-anyone with access to record can act on it**, then anyone who has access to the record will be able to act on the input prompt.
  If you select **Yes**, then you must specify one of the following options:
  - **Analyst working on the record**: Select the field that is used to assign the user corresponding to the specific module, i.e., the `People` lookup for that module. For our example, since we are working with the `Alerts` module, select **Assigned To**. You can also choose the **Created By** or **Modified By** options.
  - **Specific Users**: Users, other than the user who is assigned to the record, who require to provide the input. When you select this option, then the **People** multi-select list appears from which you can select users who require to take the decision. You can also add a custom expression in this field.
  - **Specific Team**: Team(s) who requires to provide the input, which means that any user who is part of the selected team(s) will be able to provide the input. When you select this option, then the **Team** multi-select list appears from which you can select specific teams(s) that can provide their input. You can also add a custom expression in this field.

    Important Points with respect to ownership:
    The teams or users who are specified as owners, i.e, to whom this task is assigned must have access

to the record and appropriate permissions to perform the steps required to complete the task. In case you have defined a playbook with a **Custom API Endpoint Trigger** or a **Referenced Trigger**, then an additional field **Record IRI** is added, using which you can specify the record on which the action has to be taken.

Dynamic assignment takes place in case you have selected the **Analyst working on the record** option ,i.e., the user who is currently assigned to the record is one who can take the action on the record. This means that for example, when you have executed the playbook the record is assigned to A and before A takes any action on the record, that record gets assigned to B. In such a case, B will be required to take action on the record, and A will not be able to take any action on the same.

- If you have selected the **Create a link for users outside FortiSOAR** checkbox and you require non-FortiSOAR users to provide decisions or inputs, then in the `Email Recipients` section, add a list of email addresses of non-FortiSOAR users, who should provide the required inputs or decisions using emails.

e. In the `Escalation` section, you can choose to define actions that should be taken in case a decision is not taken within the specified time-frame.
If you select **No**, then there is no time-based escalation.
If you select **Yes**, then you must specify the following:

- The time within which the action (input or decision) must be taken, in the **If the decision is not provided within** field. You can specify the time in Days, Hours, or Minutes (from version 7.0.1). The minutes option has been added for cases where responses from analysts are quickly required, such as 15-20 minutes.

- The Escalation step must be selected from the **Choose a step to continue to** field. For example, if you want to send an email notification to the managers, then you can define that step as `Escalation Email` and connect it to the Manual Input step and choose that option from the **Choose a step to continue to** field:

If you are requesting for decisions or inputs from non-FortiSOAR users via email, then you can use the escalation settings to define when the links provided in the email will expire. For example, if you select **Yes**, and specify 4 hours in the **If the decision is not provided within** field, this would mean that the links in the email that has been sent for the decision or input would expire in 4 hours.
Note: FortiSOAR runs a system schedule to resume the workflows that have timed-out, such as running the escalation step when the decision is not taken in the specified time. This schedule by default it is set to run every minute. The cron expression for this system schedule is present in the `/opt/cyops-`

`workflow/sealab/sealab/config.ini` file, and is as follows:

`MANUAL_INPUT_ESCALATION_SCHEDULE: {'minute': '*', 'hour': '*', 'day_of_week': '*', 'day_of_month': '*', 'month_of_year': '*'}`

You can update this cron expression if you want to change the default schedule timing window of 1 minute, and then run the following command:

`$ sudo -u nginx /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py default_schedules`

Also note that if the `celerybeatd` service is down then the system schedule to resume the manual input in case of an escalation step will not run. You can check the status of the celerybeatd services using the `csadm services --status` command, or by viewing the System Health Dashboard.

f. (Optional) If you want to add a link of the Manual Input dialog in the Collaboration Panel, then click the **Message** link that is present in the footer of the playbook step which displays the `Message` richtext box. In the richtext `Message` box, add the following inline code:

Inline Code Snippet: `<p><a data-comment-collaboration-pendingdecision='true' data-pendingdecision-id='{{vars.result.wfinput_id}}'>Manual Input Link</a></p>`

Important: The format of the inline code that you required to add for adding the link of the Manual Input dialog in the Collaboration Panel is changed in version 6.4.0. Therefore, if you have upgraded to a 6.4.0 or later version from a version earlier that 6.4.0, you will require to change the format of the older code snippet to match that of the new code snippet.

You can type the text that you want to display as the link text in the Collaboration Panel, which by default is set to Manual Input Link within `<a data..></a>`.

For example:

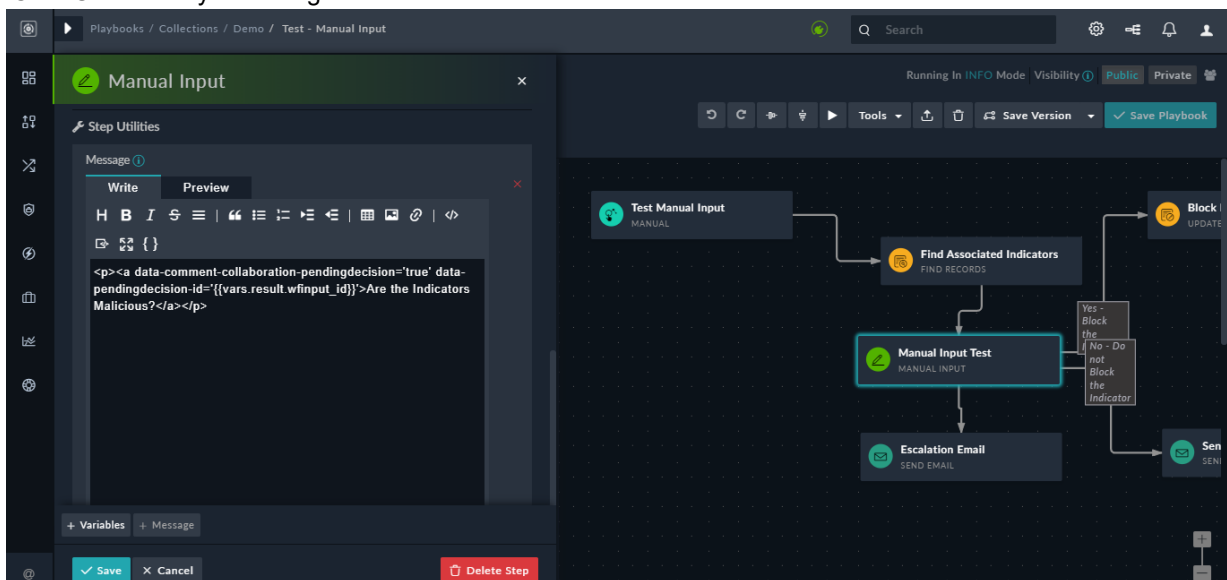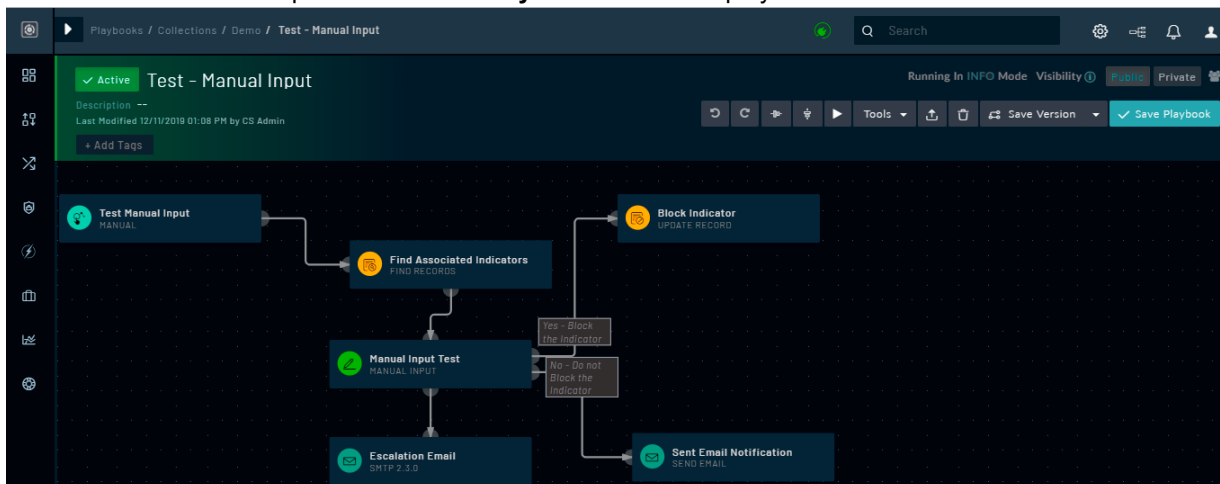*Inline Code Snippet*: `<p><a data-comment-collaboration-pendingdecision='true' data-pendingdecision-id='{{vars.result.wfinput_id}}'>Are the Indicators Malicious?</a></p>`

Click **Ok** to save your changes.

g. Click **Save** to save the step and click **Save Playbook** to save the playbook:

You can also build an input-based prompt like a decision-based prompt and you can build a prompt in the `Build Prompt - Add & define` section of the manual input step similar to the steps described in the Manual Trigger section - Building a User Prompt.

User Actions corresponding to Manual Input

The Manual input playbook gets triggered based on the type of trigger and trigger conditions defined in the playbook. For example, if you have selected a manual trigger, and then select the record that requires a manual input to be taken from a user in the module on which you have created the Manual Input playbook. For our example, we have created the Indicator Evaluation playbook on the `Alerts` module. Navigate to the **Alerts** module, then click the record for which you want to run the Indicator Evaluation playbook, and then from the **Execute** drop-down list select the **Indicator Evaluation** action to trigger the Indicator Evaluation playbook as shown in the following image:
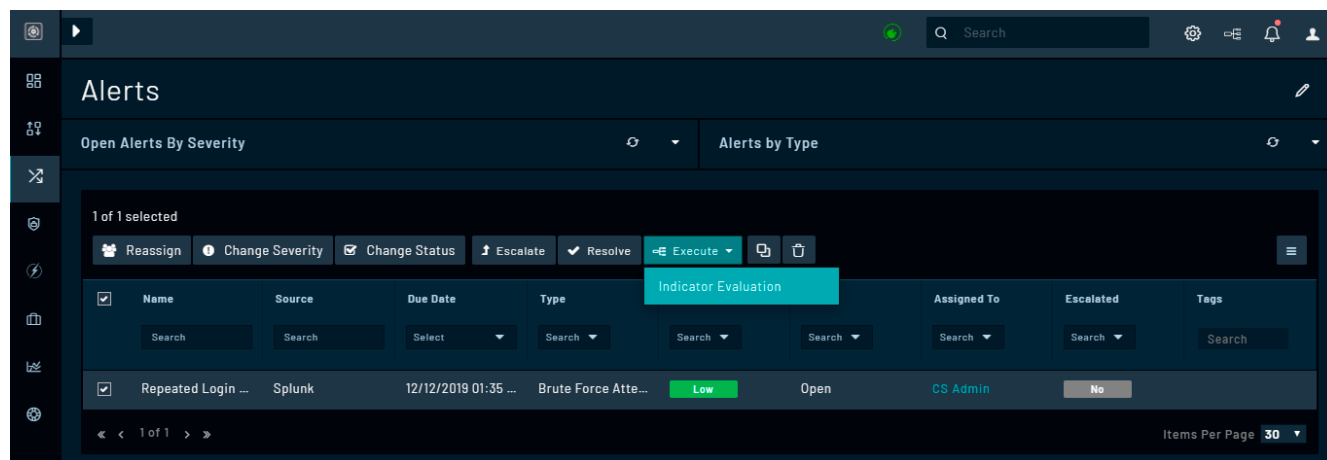
Once you trigger the Indicator Evaluation playbook, FortiSOAR does the following:

- Displays a message such as `Triggered action "Indicator Evaluation" on 1 record` and halts the further execution of the Test - Manual Input Playbook. You can open the **Executed Playbook Logs** by clicking the **Executed Playbook Logs** icon in the upper right corner of the FortiSOAR. You will see that the status of the `Test - Manual Input` Playbook is set to **Awaiting** as shown in the following image:
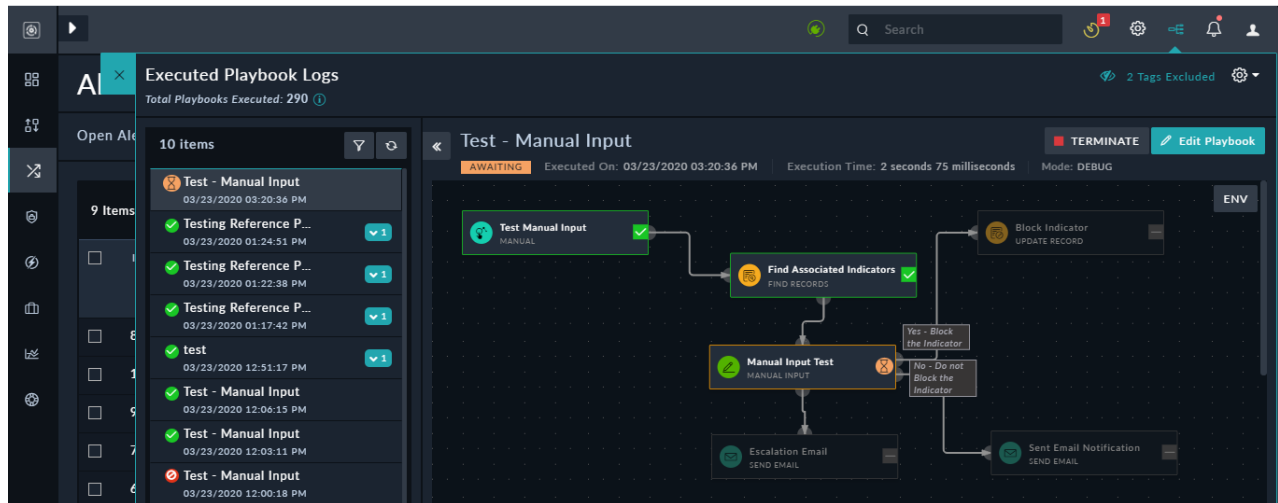
- If the decision or inputs are required to be provided by a FortiSOAR user, then users have to do the following:
  - Users can either use the manual input link that appears in the Collaboration Panel or the Pending Decisions button (described later in the section) to display the Manual Input dialog. Users must add use the Message action and add a message to their playbook to add a link to the Manual Input dialog as described in the Building a decision-based input prompt procedure. Clicking the link in the collaboration panel will open the pending decisions dialog in which users can provide your input, which would then resume the playbook workflow, which has been described later in this section.
  - A **Pending Decisions** icon appears on the top-right corner in FortiSOAR when a decision is pending. This button contains a number that indicates the number of inputs or decisions that are pending, the number appears in the red color. Click the **Pending Decisions** icon to see its details as shown in the following image:
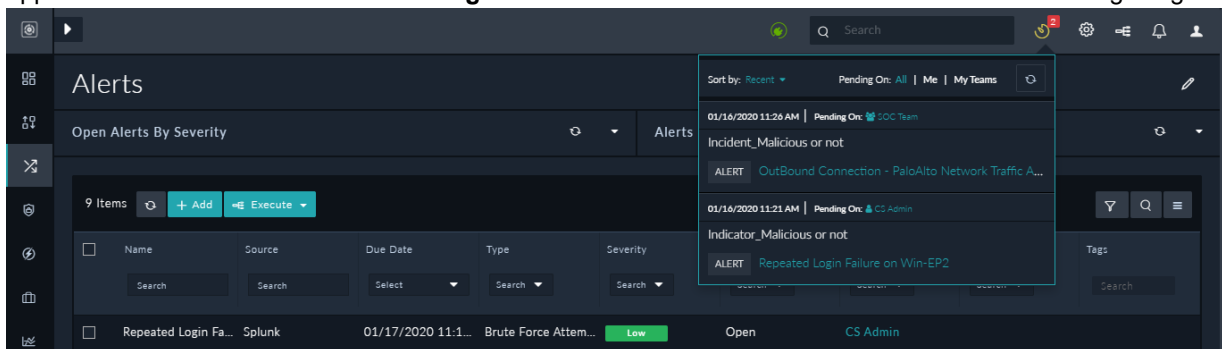
  The Pending Decisions list contains details such as created date, the person or team the decision is assigned to, title of the manual input step, type of record on which the action is pending, for example `Alert` as shown in the above image, and due date till when the decision should be taken are displayed. Users can also *sort* the pending decisions by **Recent**, i.e., based on its created date or on the **Due By**, which is the date by which a decision requires to be given. You can *filter* pending actions list by **All**, which displays all the pending items, **Me**, which displays the pending actions that has been to **current user**, or **My Teams**, which displays the pending actions that has been assigned to the team(s) of the current user.
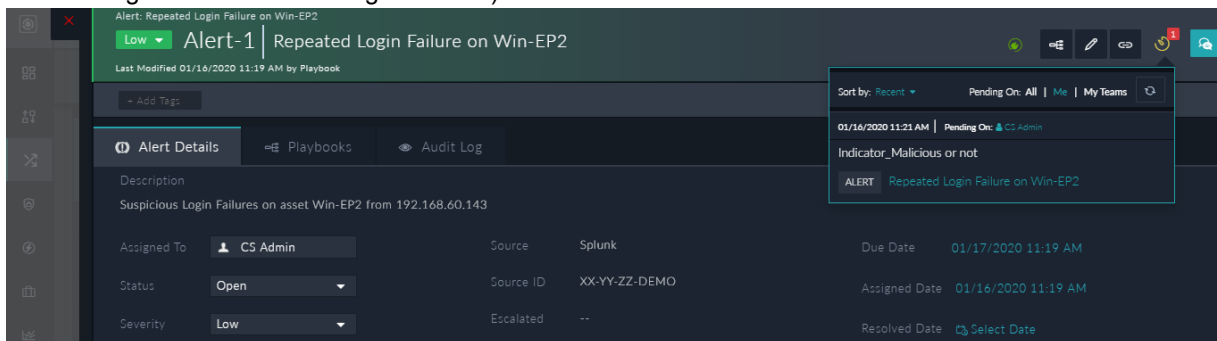 In the **Pending Decisions List**, click the item to provide an input, which displays a `Pending Details` decision box (popup) in which the user can add their input as shown in the following image:
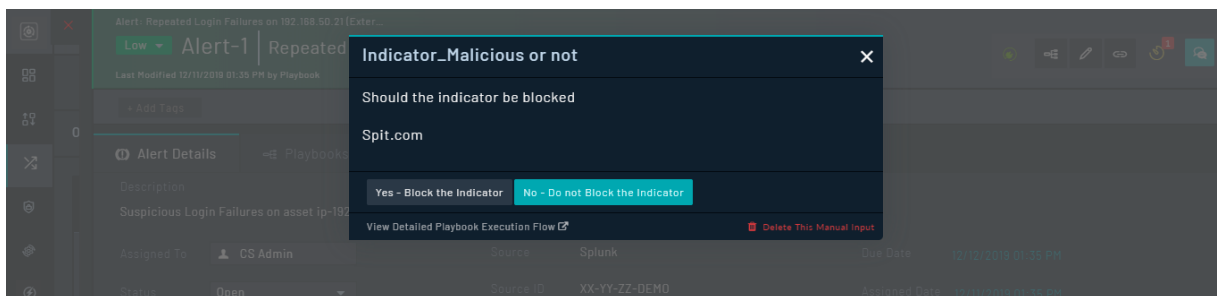
If you click the **View Detailed Playbook Execution Flow** link, a new window opens that displays the execution of the playbook based on the input or decision received. Also, as you can see in the above image, since in the playbook selected **No - Do not Block the Indicator** has been specified as the *Primary* action, that option gets highlighted in the popup.

Users can use the **Delete This Manual Input** link to discard the manual input and remove this input from playbook workflows or queues. This is required to discard manual inputs without references. For example, in a playbook that contains manual input, manual task, or approval step, and a step after the manual input, manual task, or approval step fails, then the complete playbook is marked as failed; however, the manual inputs are still open for user action even after the awaiting steps are terminated. Therefore, you can use **Delete This Manual Input** to completely discard the manual input. Other examples would be in cases where an executed log entry is removed without addressing the open manual input requests or deletion of a record that requires a manual input.

- You can also see the **Pending Decisions** icon in the detail view of the alert on which the playbook is triggered and depending on the ownership you have defined in the playbook. For example, if you have provided the ownership of **Analyst working on the record**, and if you are not assigned to that record, then you will not see the Pending Decisions button. The details of the Pending Decisions list is the same as the details displayed in the Pending Decisions list at the global level) :

Clicking the item in the **Pending Decisions** list displays the `Pending Details` decision box as explained earlier.

- If, for example, the user selects the **Yes - Block the Indicator** option in the `Pending Details` decision box, then FortiSOAR displays a message such as `Awaiting playbook resumed successfully` and based on the user's decision, FortiSOAR continues the execution of the playbook. For our example, the 'Block Indicators' step will be run that will block the indicator. Users can open the **Executed Playbook Logs** by clicking the **Executed Playbook Logs** icon in the upper right corner of the FortiSOAR, and there they will see that the status of the "Test - Manual Input Playbook" is set to **Finished**, the Block Indicator step is executed, and the Escalation Email and Send Email steps are skipped, as shown in the following image:
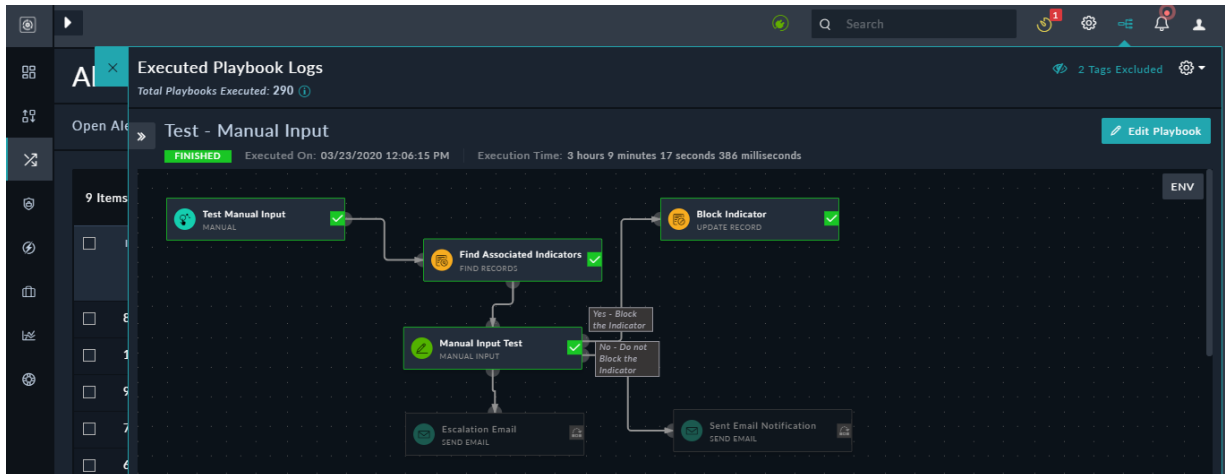
An analyst or user on whom the action is awaiting can also provide the input from the **Executed Playbook Logs**. Click the **Executed Playbook Logs** icon in the upper right corner of the FortiSOAR to open the Executed Playbook Logs and click the playbook whose status is **Awaiting**. Clicking the awaiting playbook, opens the **Test - Manual Input** playbook > **Pending Inputs** tab on the right side of the `Executed Playbook Logs` dialog in which you can add and submit your inputs as shown in the following image:
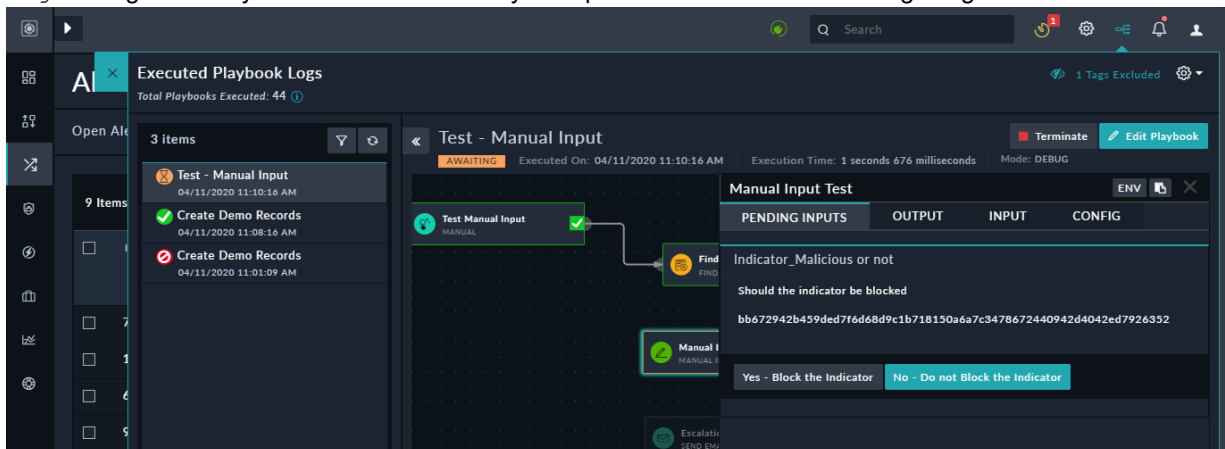
- If the decision or inputs are required to be provided by a non-FortiSOAR user via email, then users have to do the following:
  - Once the playbook is triggered and the playbook is set to **Awaiting**, an email gets sent to the email addresses mentioned in the playbook. The email body contains text such as, "A FortiSOAR Playbook is requesting your input..."and a link such as, "Open input form".
    You can customize the text of the email body by customizing the email template (*System: Send email for non-FSR user manual input*) that is present in **Resources** > **Email Templates**.
  - Clicking the link opens the browser and displays a page in which users require to provide your input or decision. The contents of this page depends on the title and description that you have added in the playbook, along with the two buttons for acceptance or rejection of the decision, if it is a decision-based prompt. In case of our example, the user will see the "Should the indicator be blocked" followed by the indicator value and then two buttons "Yes - Block the indicator", or "No - Do not block the indicator".
    If the prompt is a input-based prompt, then users will see an 'Input Form' containing fields that have been defined in the playbook. Users should provide the necessary inputs and then submit the form. Once the form is submitted, it cannot be re-opened and its contents cannot be changed.

Once the user provides the required inputs and submits their action, the playbook continues its execution as per the defined workflow.

Execute

Connector

Use the **Connector** step to add connectors, including FortiSOAR Built-in connectors, to your playbook. Third-Party Connectors, such as connectors for Elastic, VirusTotal, or Splunk, can retrieve data from custom sources and perform automated operations. FortiSOAR Built-in connectors, such as the Database connector, the IMAP connector, and the SMTP, are all pre-installed connectors or built-ins that you can use within FortiSOAR playbook and perform automated operations. For more information on FortiSOAR Built-in connectors, see the "FortiSOAR Built-in connectors" article.

Use the **By Connector Name** tab to first choose a specific connector and then choose the operation that you want that connector to perform or use the **By Actions** tab to first choose the action (annotation) that you want to perform and then choose the connector that you want to use to perform the selected action.

Once you click the **Connectors** step, the `Connectors` step page is displayed that contains the connectors (**By Connector Names** tab) that are configured in your system and the automated actions that you can perform (**By Actions** tab).
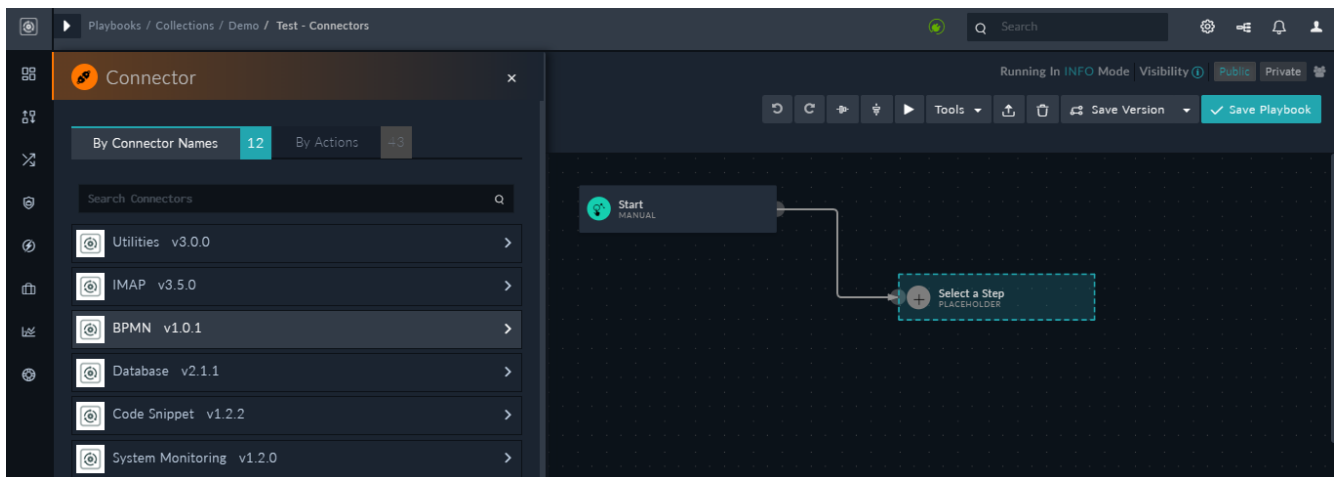
By Connector Names tab

After selecting the **Connector** step in the playbook designer, the **By Connector Names** tab is displayed. The **By Connector Names** tab displays all the connectors that are configured in your system. Use this tab if you want to use a particular connector to perform a particular action.
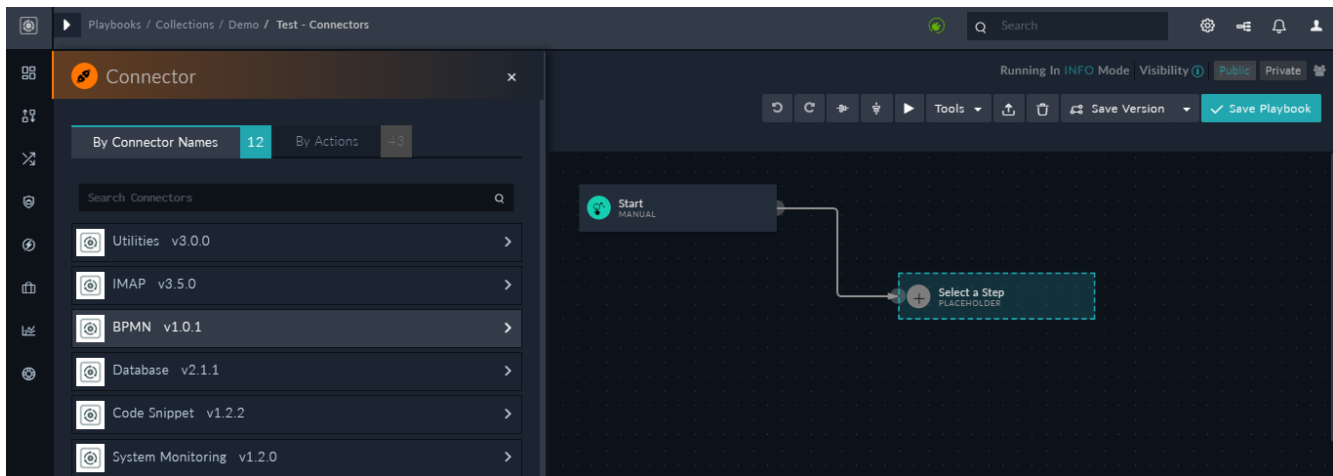
Use the `Search Connectors` section to search for connectors by name.

Click the connector that you want to include in your playbook, for example, **VirusTotal v1.0.1** and then type the **Step Name**. You can also specify whether you want to run the action on the current FortiSOAR node or remotely on the agent node by clicking the **Self** or **Agent** buttons besides `Target`. By default, **Self** is selected, which means that the action will run on the current FortiSOAR node, then you must select the configuration by clicking the **Configurations** drop-down list using which you want to run the action since the FortiSOAR node can have multiple configurations. Configurations are based on the configuration names that you specify when you are configuring the connector (see notes below). If you click **Agent**, then you can select the agent on which you want to run the action and you must also select the configuration using which you want to run the action since agents can have multiple configurations. For more information on agents and how to run remote actions using agents, see the *Segmented Network support in FortiSOAR* chapter in the "Administration Guide." You can also specify the connector configuration by clicking the **{}** icon and either typing the connector configuration name or specifying a Jinja variable that contains the connector configuration name. If you have only one configuration for the connector or have specified a default configuration, then that configuration automatically gets selected.

Users can see only those connector configurations to which they have access. For example, if a VirusTotal connector is configured with configuration name as 'Demo1' and with visibility set to 'Private' with assignment given to 'Team 1' (for more information on playbook ownership, see Introduction to Playbooks chapter), then the 'Demo 1' configuration is not visible to users belonging to teams other than 'Team 1', though they can execute playbooks created by 'Team 1' users.

Next, from the **Action** drop-down list, select the action that you want the connector to perform and then in the `Inputs` section, specify the inputs required. Click **Save** to add the connector as a playbook step.
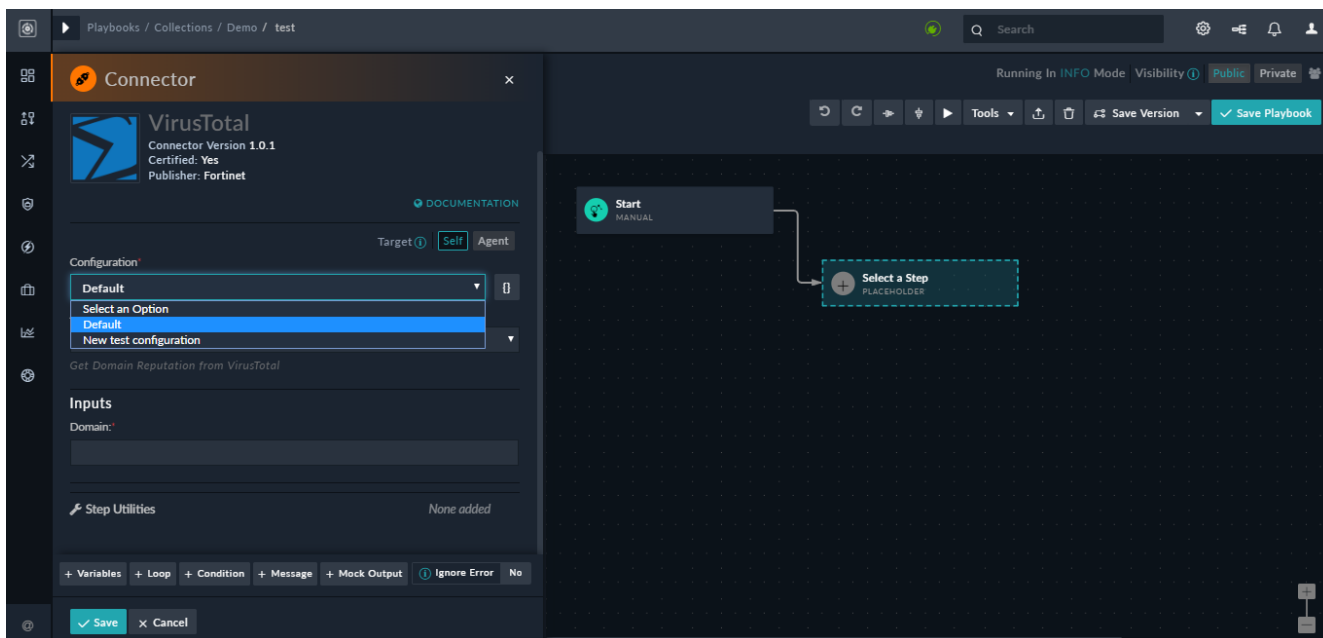
Notes:

- You can install different versions of a connector, and while adding a connector operation, you can specify a specific version of a connector within a playbook. For example, you can have VirusTotal connector versions 1.0.0 and 1.0.1. The version of the connector must be in the **x.y.z** format, for example, **1.0.0**. Version must consist of valid integers, for example, "1.15.125" is a valid version.
  In case you have installed multiple connectors, and if the version of the connector specified in the playbook is not found, then the playbook by default uses the latest version. FortiSOAR checks for the latest version of the of the connector in the format "major version.minor version.patch version". For example, version 2.0.1 is a later version than 1.0.1.

- Upgraded versions of your connector are displayed on the Connectors page and you can upgrade the version of your connector. The upgrade process replaces your existing connector version with the upgraded version. For more information, see the *Introduction to connectors* chapter in the "Connectors Guide."

- You can install different versions of a connector, enabling you to reference a specific version of a connector from a playbook. If you want to replace all previous versions of the connector, ensure that you click the **Delete all existing versions** checkbox while importing the new version of the connector. If you do not click the **Delete all existing versions** checkbox, then a new version of the connector is added. You must ensure that your playbooks reference a correct and existing version of the connector.

- You can add multiple configurations for your connector if you have more than one instance of your third-party server in your environment. You must, therefore, add a unique `Name` for each configuration.
  If you have previous versions of a connector and you are configuring a newer version of that connector with the same configuration parameters, then FortiSOAR fetches the configuration and input parameters of the latest available version of that connector. For example, if you have 1.0.0 and 1.0.1 versions of the VirusTotal connector and you are configuring the 1.0.1 version of the VirusTotal connector, then while configuring the 1.0.1 version, FortiSOAR will fetch the configuration and input parameters from the 1.0.0 version of the VirusTotal connector. You can review the configuration and input parameters, and then decide to change them or leave them unchanged.

- You can check the **Mark As Default Configuration** option to make the selected configuration, the default configuration of this connector, on the particular FortiSOAR instance. This connector will point to this configuration by default.

- The `password` type fields include encryption and decryption. All configuration fields of type `password` are encrypted before they are saved in the database.

By Actions tab

After selecting the **Connector** step in the playbook designer, if you want to see the available connectors configured in your system for a particular action, then click the **By Actions** tab. Click the **down** arrow to view which connector is providing that action and the description of the action. The **By Action** tab displays the connectors grouped by actions.
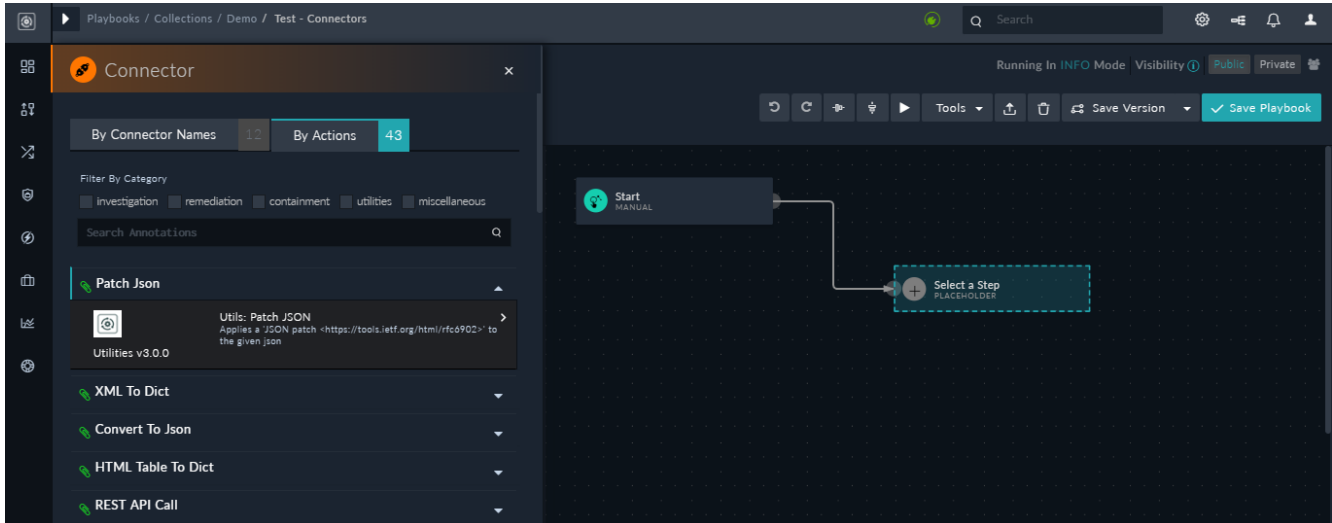
Use the `Filter By Category` section to filter the actions on the basis of the type of operation they will perform. The types of operations are currently categorized into **Investigation**, **Remediation**, **Containment**, **Utilities**, and **Miscellaneous** categories.

To search for a specific action that you want to perform, type the search keyword in the **Search Annotations** search box.
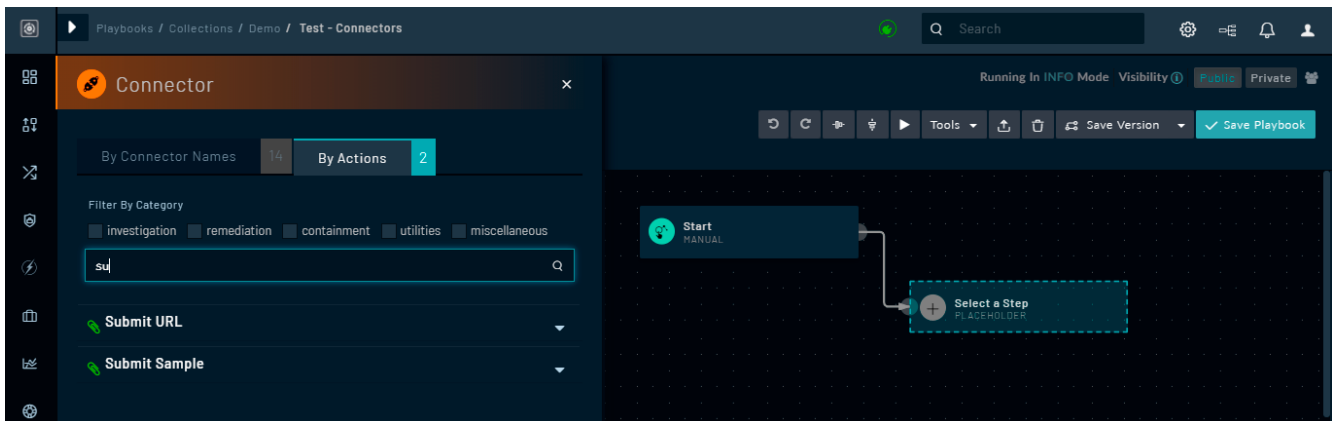
Click the name of the action that you want to perform. For example, if you want to submit a sample for analysis to a website or a sandbox click the **Submit Sample** action. Once you select the action, then a list of configured connectors that can perform that operation is displayed as shown in the following image:
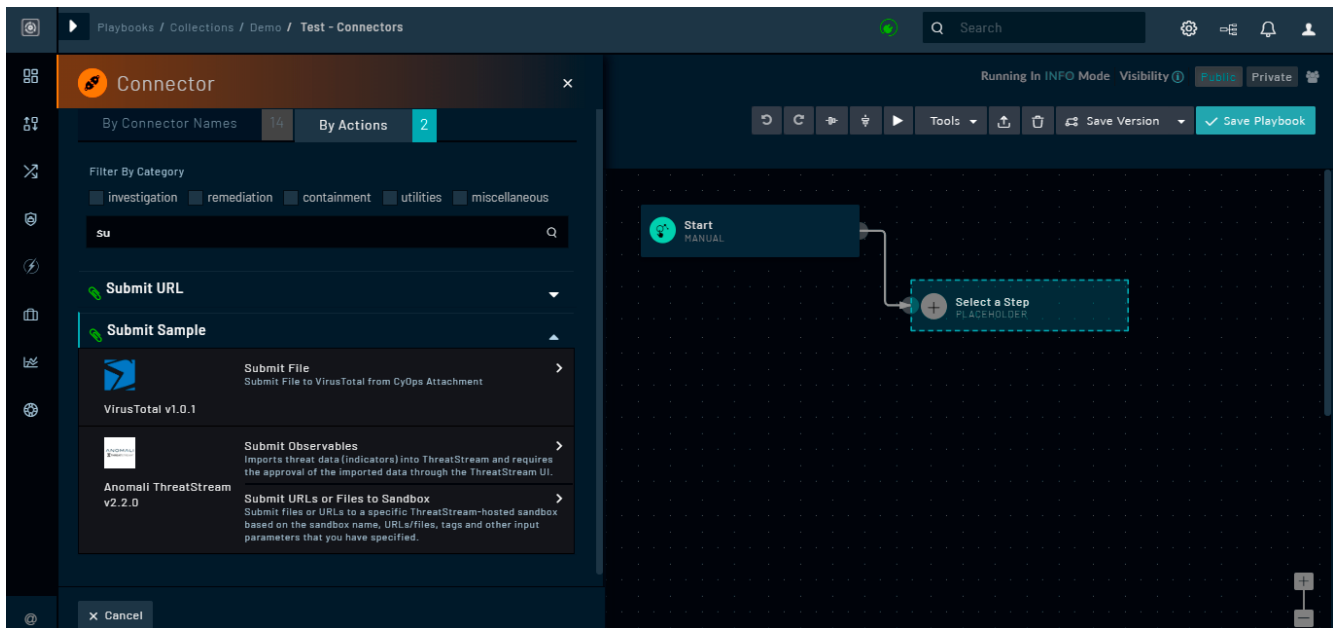
An annotation can have multiple connectors configured to perform that action, and if more than one connector can perform the same action, then a list of connectors will be displayed when you click the name of the action. As in our example, we want to submit a sample for analysis click the **Get Sample** action, and you will see that multiple connectors, such as `VirusTotal` and `Anomali ThreatStream` are tagged with this annotation.

Select the connector and the exact operation that you want to perform and then type the **Step Name**. Next, in the `Inputs` section specify the necessary input parameters to run the operation. Click **Save** to add the connector as a playbook step.

For example, to submit a sample for analysis click the **Submit Sample** action and you will see the connectors associated with this action. Select the connector, for example, the `Anomali Threatstream` connector, and you will see multiple functions, such as **Submit Observables** and **Submit URLs or Files to Sandbox**, associated with the desired action as shown in the above image. Click the exact operation that you want to perform, for example, if you want to submit files or URLs to a specific ThreatStream-hosted sandbox, then click **Submit URLs or Files to Sandbox**. Next, type the **Step Name**, and in the `Inputs` section, enter the input parameters, such as the sandbox name and sample type that you want to submit for analysis to Threatstream, and then click **Save** to add the connector as a playbook step.
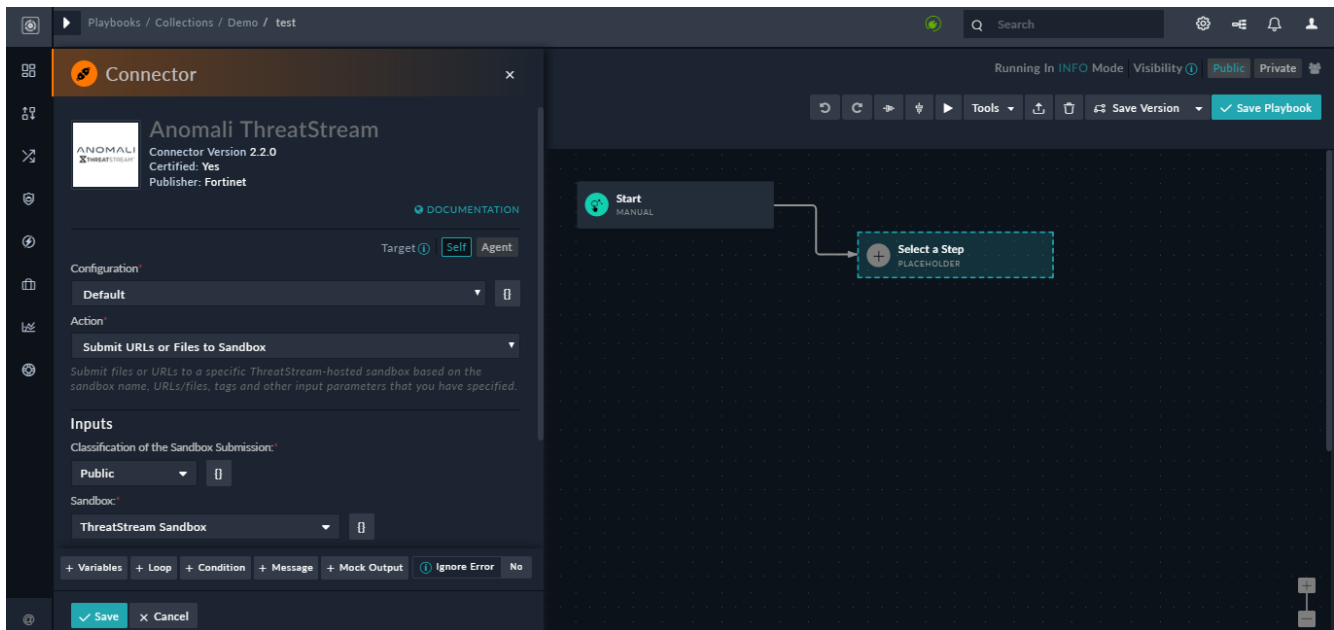
In case of connector actions that have the `Datetime` field, you can use the Date and Time Picker to choose the date as shown in the following image:
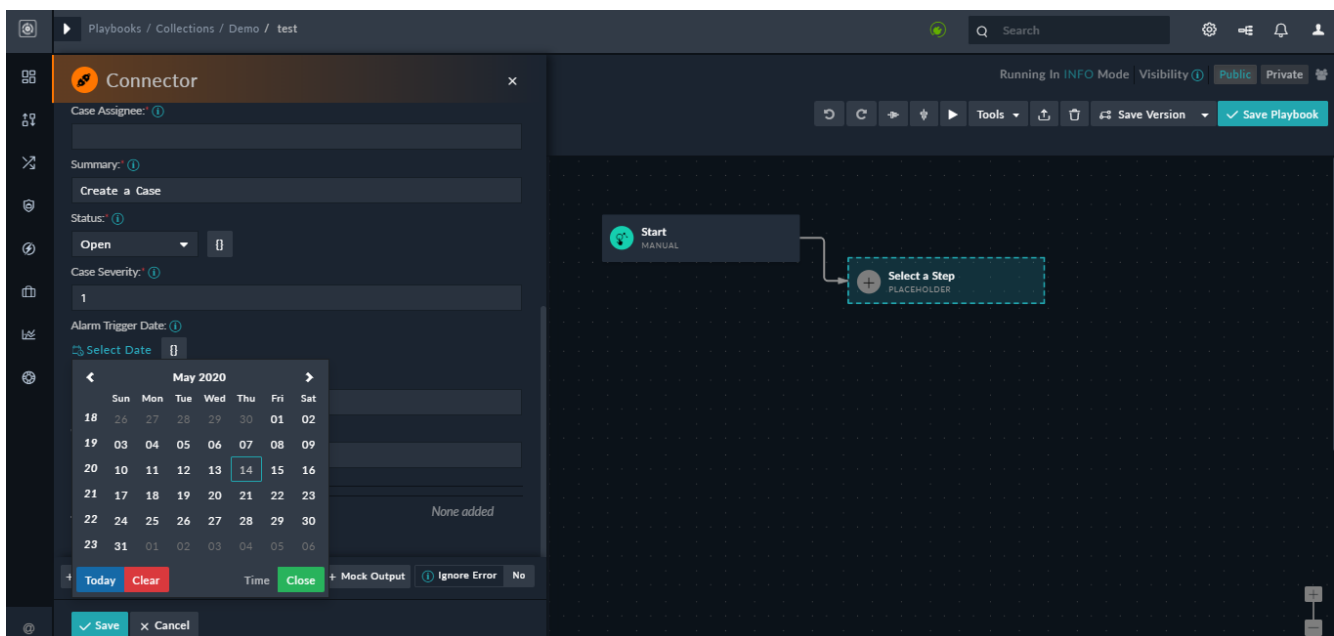
You can also add custom expressions in the jinja format in the `Datetime` field. Click the ⬜ icon to enter Jinja for this field. Click the ◀ icon to toggle back to the original `Datetime` field.

Utilities

Use the **Utilities** step to run various utility functions and scripts that come built-in with FortiSOAR.

Utility functions include functions such as, the **Utils: Make REST API Call** option to make a RESTful API call to any valid URL endpoint, the **FSR: Create Record** option to insert a new record in FortiSOAR, and the **File: Zip** option to zip and password protect a file.

Example of using the FSR: Upsert Record option in the Utilities step

The **FSR: Upsert Record** step either updates an existing record, if any record matches the unique list of fields you have specified, in the database, or inserts a new record in the database based on the parameters you have specified.

Upsert behavior for uniqueness will not work for fields that are marked as encrypted.

In the Playbook Designer, click the **Utilities** step and add the step name in the **Step Name** field. From the **Action** drop-down list, select **FSR: Upsert Record**. In the `IRI` field add the name of the module in which you want to upsert data in the format `api/3/alerts`. In the `Body` field, add the fields that you want to add or update in the database in the *dynamic values* (Jinja variables) format. For example, `{ "name" : "alert1", "description" : "Test for Upsert", "status" : "{{"AlertStatus" | picklist("Open", "@id")}}"}`. In the `Fields` field, add the list of fields to check for uniqueness. For example, if you want to check for records in the database based on the `Name` of the record in the database, add `['name']` in the `Fields` field. If you want to search the database based on multiple items, you can add more than one item in the `Fields` field, for example, `['name', 'status']`. The `Ignore Missing Fields` field is used to determine whether or not to raise an exception if you specify a field in the `Fields` field that is not in the record. The `Ignore Missing Fields` defaults to `False`, which means that an exception will be raised if you specify a field in the `Fields` field that is not in the record. Click **Save** to save the step.
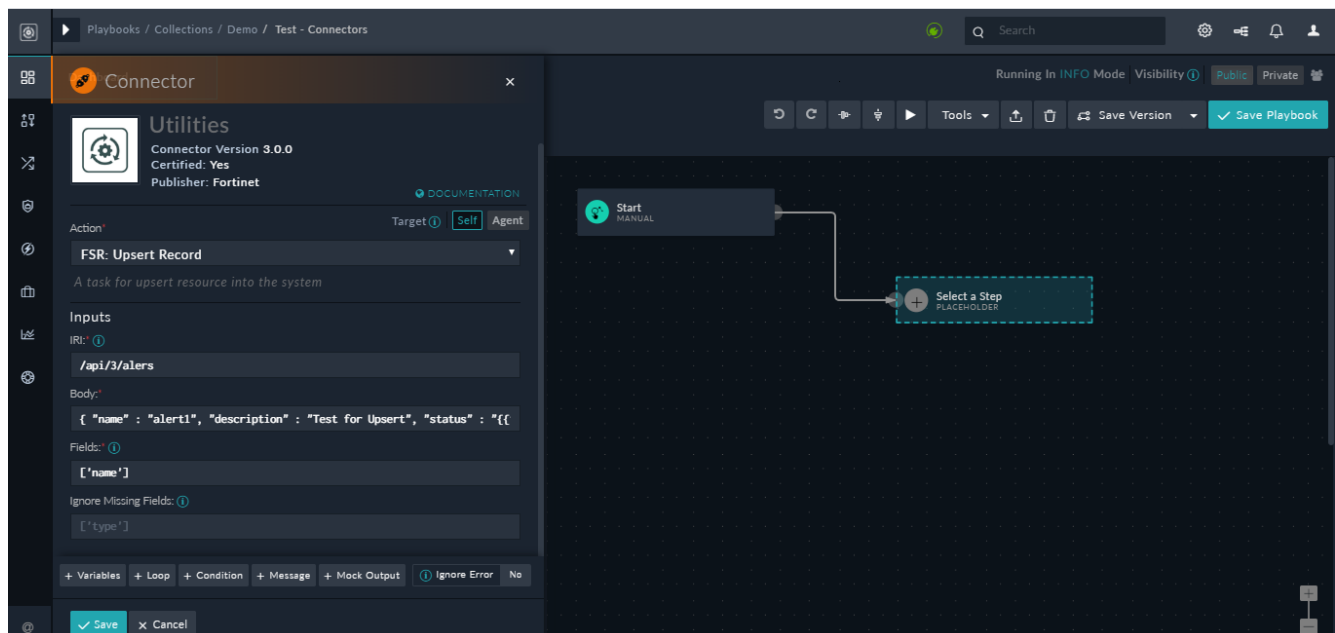
Once the step is run, the database record will either be updated with the parameter you have specified, if any record matches the list of fields you have specified in the `Fields` field, or a new record will be inserted in the database based on the parameters you have specified.

Execute Code Snippet

Use the `Execute Code Snippet` step to run a python function as part of the playbook.

This step has only one action which is **Execute Python Code**, so select this option and in the **Python Function** field enter the python function that you want to run as part of the playbook and then click **Save** to save the step.

This step uses the Code Snippet connector as its base, for more information on FortiSOAR Built-in connectors, including the Code Snippet connector, see the "FortiSOAR Built-in connectors" article.

References

Reference a Playbook

Use the **Reference a Playbook** step to call any playbook within the system, whether Active or Inactive, by name. A child playbook can reference all environment data from its parent playbook, meaning if a child playbook requires a particular dynamic value, the child playbook can reference that variable and used it just as it is being used in the parent playbook.

To add a reference to a playbook, click the **Reference a Playbook** step and in the **Step Name** field, type the name of the step, then in the `Playbook Reference` field, click **Select**, which displays the `playbookReference` list. The `playbookReference` list displays a list of all the available playbooks across the playbook collections from which you can select the playbook that you want to reference. You can also click the **Add Custom Expression** (▯) button beside **Select** to specify jinja variable that contains the IRI value of the reference playbook.

You can use the **Loop** option to iterate a playbook step as per your requirements.

💡 If you have migrated a Map Playbook to a Reference a Playbook (using Loop), you observe a change in behavior. In the case of Map Playbook, any changes done to the environment variables by the Map Playbook were reflected in the Main Playbook directly. However, in the case of Reference a Playbook, you must explicitly set the returned values from the referenced (child) playbooks in its last step. This ensures that the child playbook does not change the behavior of the main playbook in an unexpected manner.

The output of the reference playbook steps varies depending on the called playbook parameters. You can define parameters using **Tools** > **Edit Parameters** in the playbook designer
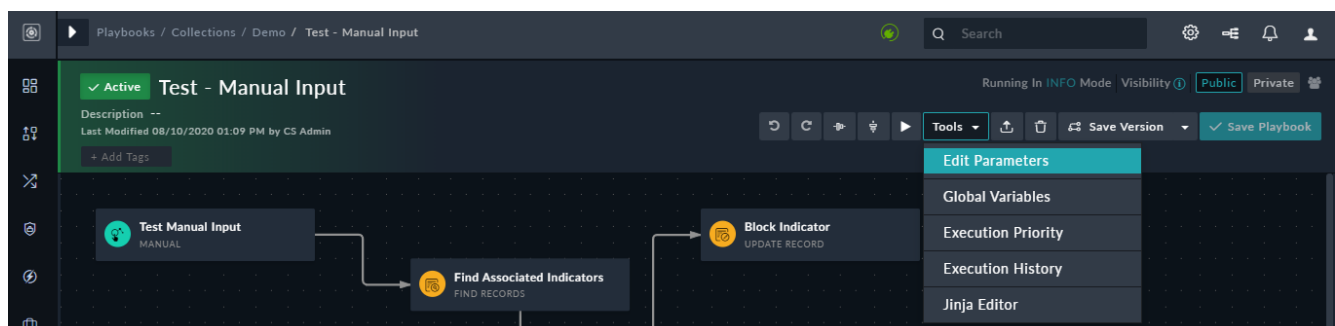
> If you update any of the parameters in a child playbook, then you must review and make the necessary updates in the Reference a Playbook step in the parent playbook. For example, if you have deleted a parameter from a child playbook, the parameter will yet pass from the parent playbook to the child playbook since the input values are saved in the reference playbook step. These inputs are cleared only when you open and save the Reference a Playbook step in the parent playbook.

If you want to use a variable from a playbook that your are referencing (A) in the calling playbook (B), then defined that parameter in the referenced playbook (A) using **Tools** > **Edit Parameters**. This is the recommended method of passing environment variables from the referencing (parent) playbook to the referenced (child) playbook. It is *not* recommended to directly use the environment variable (since the parent environment is available in the child workflow as well) without explicitly defining child playbook input. You can turn this feature (passing of parent environment variables) **on** or **off** by updating the following entry in the `celeryd` section of the `/opt/cyops-workflow/sealab/sealab/config.ini` file:

`COPY_ENV_FOR_REFERENCE_WORKFLOW : false`

Restart the FortiSOAR services once you have updated the entry in the `config.ini` file.

By default, the `COPY_ENV_FOR_REFERENCE_WORKFLOW` is set to `false`.

Email

Send Email

Use the **Send Email** step to create a step that will prompt the executed playbook to automatically send an email to the user(s) identified in the step with either specific static criteria or record-relevant data using dynamic values.

If the email needs to reflect data specific to the entity that triggered the playbook, then use *dynamic values* in the fields.

Following are some examples of how you can send an email with jinja content In case of On Create or On Update triggers:

- To send an email to the user who is assigned to a Task, enter the following in the *TO* field:
  `{{vars.input.records[0].assignedTo.email}}`.
- To set the email subject line as the name of the Task/Incident, enter the following in the *Subject* field:
  `{{vars.input.records[0].name}}`

If the email will always have the same recipient/content/etc., then enter the text in the corresponding fields and click **Save**.

> If you have stored a comma-separated list of multiple email addresses in any 'Set Variable' step and you use that jinja variable in the 'TO' field in the 'Send Email' playbook step then the email is not sent to all the email addresses. If you require to send the email to multiple email addresses, you must use the FortiSOAR provided SMTP built-in connector to add multiple email addresses in the 'TO' field. The SMTP connector has a `Send Email` function that supports multiple addresses both in the jinja variable and string formats.

The **Send Email** step uses the SMTP Built-in connector and you can send emails to existing FortiSOAR teams or users by selecting teams or users from pre-populated multi-select fields. For more information on FortiSOAR Built-in connectors, including the SMTP connector, see the "FortiSOAR Built-in connectors" article.

The **Send Email** step has deprecated the previous *Send Email* and *Send Rich Text Email* steps have been deprecated and have been replaced with the *Send Email* step. The **Send Email** step that has been enhanced to send a rich text email with jinja and email template support. In the **Actions** drop-down list, however, to support backward compatibility, you will see all three options, **Send Email** and **Send Email (Deprecated)** and **Send Rich Text Email (Deprecated)**:
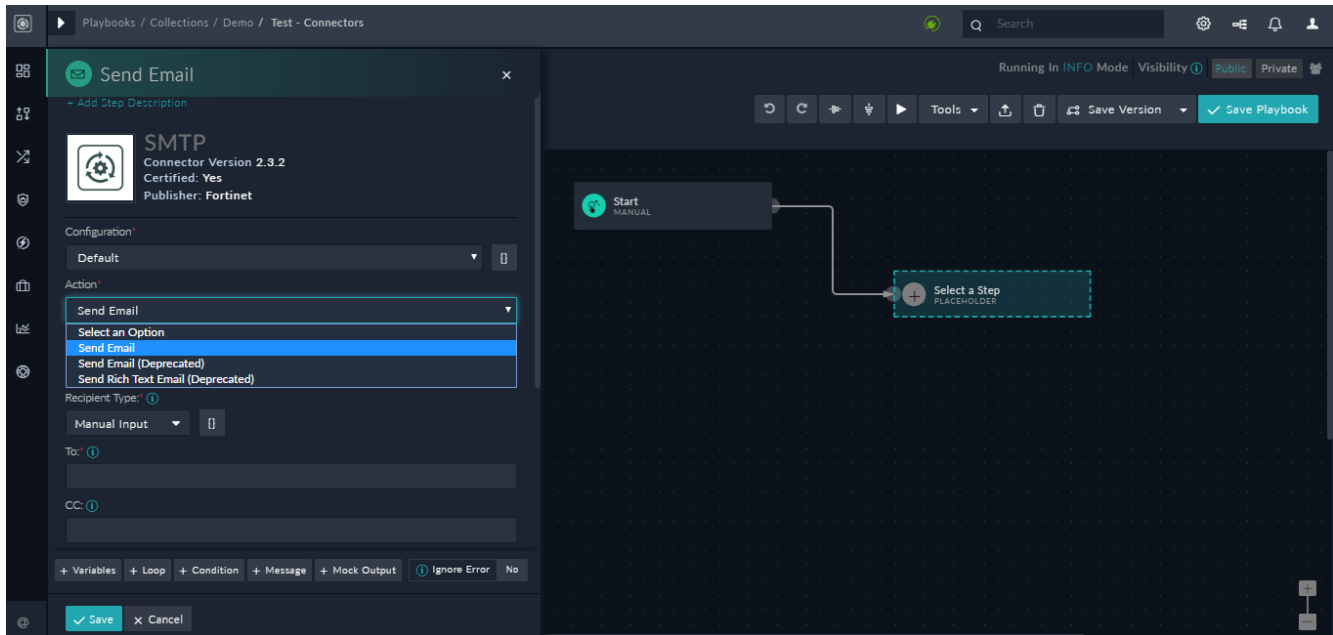
The `Send Email` step provides user with the ability to pass an existing email template as an input for the email subject and body (content), thereby, enabling users to leverage an existing template and build upon it, and therefore, avoid re-work and ensuring consistency. The newly added **Send Email** step contains a **Body Type** drop-down list from which you can choose whether you want to send a plain text email (**Plain Text**), rich text email (**Rich Text**), or an email based on a template (**Email Template**):
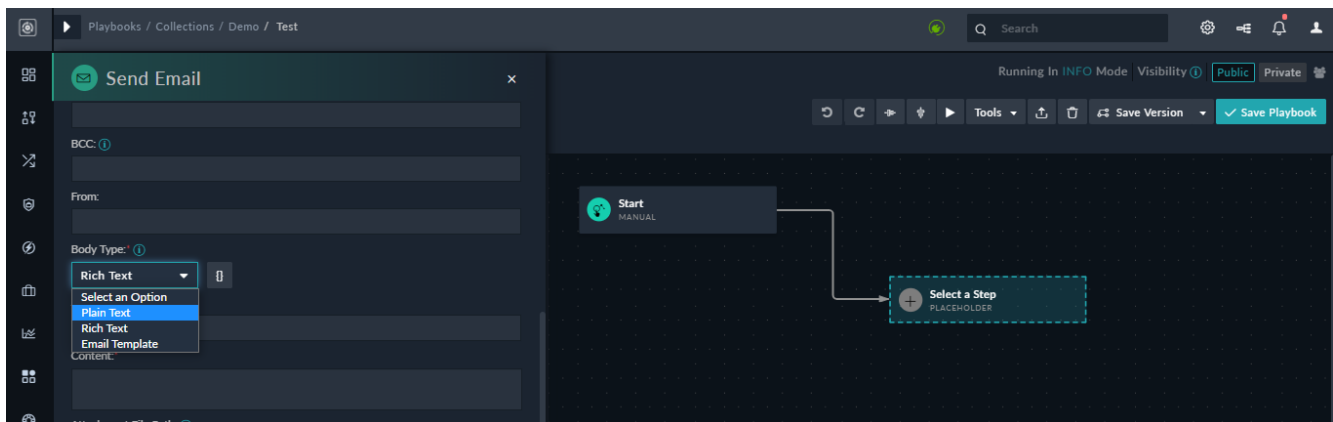
If you select **Rich Text** from the **Body Type** drop-down list, then in the **Content** field, you can add formatted content, images, and even custom jinja expressions using Dynamic Values.

If you select **Email Template** from the **Body Type** drop-down list, the **Email Template** drop-down list gets displayed, using which you can select the template that you want to use to send the email:
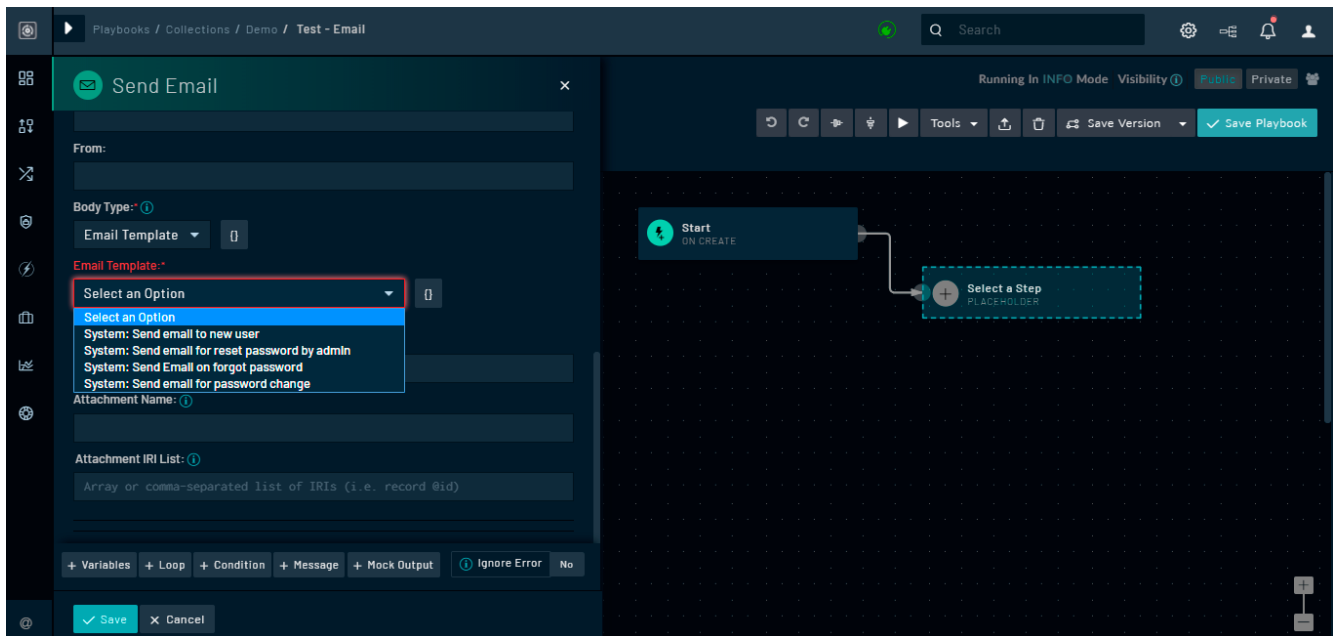
Authentication

Set API Keys

You can change the context of the user, i.e., override the default appliance keys using the `Set API Keys` step. For a particular playbook if you wish to run the API steps with less or more privileges than that of the default Playbook appliance, you can do so by adding the `Set API Key` step before the concerned steps in the playbook. In this case the privileges of the specified API key will be used; and this will apply to all steps in the playbook after the `Set API Key` step.

You can also use the `Set API Keys` step to create a playbook using the no authentication webhook (No Authentication trigger) in case of the Custom API Endpoint Trigger. In such a case, to successfully perform any operation, such as creating a record in FortiSOAR, you will require to use the Set API Keys step and provide the appliance keys for authentication.

To use the `Set API Keys` step, open the playbook and click the **Set API Keys** step and in the **Step Name** field, type the name of the step. Next, enter the **Public Key** and **Private Key** values and click **Save**. For details on generating a public and private key, or retrieving the details of a public key, see the *Appliances* topic in the "Security Management" chapter in the "Administration Guide."

The owner of the records created or updated by this playbook are the teams who own the appliance whose keys are specified in the playbook.

Deprecated Playbook steps and triggers

If you are using a deprecated step or trigger in a playbook, in cases where you have upgraded from an older version of FortiSOAR, then that playbook will continue to work, and you can edit the deprecated step. In case of deprecated steps, FortiSOAR displays a message such as "`This step is deprecated.....`"

⚠️	If you are using deprecated steps or triggers in your playbook, it is highly recommended that you replace those steps and triggers because over time these steps and triggers will become obsolete, and FortiSOAR will not be able to support or respond to them. You can replace the deprecated steps with the **Utilities** step or by using the FortiSOAR Built-in connectors. For more information on FortiSOAR Built-in connectors, see the "FortiSOAR Built-in connectors" article.

Deprecated Playbook Triggers

Pre-Data Operation Triggers

Pre-data operations have been deprecated and they are intended for synchronous operations, where the data operations might potentially block or affect the final data updates to the database. Therefore, pre-data operation triggers perform some action before the data operation is completed in the database.

Example of a pre-data operation trigger: Suppose your organization has an allowlist database and you want to ensure that before an alert is created its IP address is checked against the database. If the IP address is part of the allowlist database, you do not want an alert to be created.

The following table lists the types of Pre-Data Operations triggers that have been deprecated:

Deprecated Playbook Trigger Name	Brief description of the trigger
Pre-Create	This trigger starts the execution of a playbook immediately before inserting the selected model type to the database. You can create a Pre-Create trigger on almost all models, which includes Modules.
Pre-Update	This trigger starts the execution of a playbook immediately before updating the selected model type to the database. You can create a Pre-Update trigger on almost all models, which includes Modules. You add a Pre-Update trigger in the same way you added a Pre-Create trigger.
Pre-Delete	This trigger starts the execution of a playbook immediately before deleting the selected model type to the database. You can create a Pre-Delete trigger on almost all models, which includes Modules. You add a Pre-Delete trigger in the same way you added a Pre-Create trigger.

Deprecated Playbook Steps

The following playbook steps have been deprecated from version 4.11 and later since most of them have been added to the **Utilities** step and as part of FortiSOAR Built-in connectors.

> If you are using deprecated steps in your playbook, it is highly recommended that you replace those steps with the **Utilities** step or by using the FortiSOAR Built-in connectors since over time these steps will become obsolete and FortiSOAR will not support them.

The following table lists the steps that have been depreciated and the step or connector that you can use instead of the deprecated step:

Deprecated Step Name	Step or con-nector that replaces the deprecated step	Brief description of the step
Add Database Connector	Database Connector	To connect to a particular database.
Run Script	Utilities Connector	To run various scripts.
Database Query	Database Connector	To query a database to which you have established a connection.
Remote Command	SSH Connector	To connect to a remote machine and execute the required commands.
SFTP	Utilities Connector: `uploadfile url` operation	To connect to a particular SFTP URL.
Make API Call	Utilities Connector	To make a RESTful API call to any valid URL endpoint.
Fetch Email	IMAP Connector	To retrieve an email from a specified host.
Create File from String	Utilities Connector: `create file from string` operation	To create a file from a string input.
Download File from URL	Utilities Connector: `download file from URL` operation	To download a file from a particular URL.
Create Attachment from File	Utilities Connector: `create attachment from file` operation	To add a file to the Attachments module within FortiSOAR.
Map Playbook	Reference a Playbook step	To call any playbook within the system using the IRI of the playbook
Run Utility Functions	Utilities Connector	To run various utility functions.
Pause	Wait	To pause the execution of a playbook step. **Note**: The support for the `Pause` step has been completely removed. You

		must use the `Wait` step.
Manual Decision	Manual Input	To pause the execution of the playbook until the user or analyst who is assigned to make the decision provides the choice. to pause the execution of the playbook until the user or analyst who is assigned to make the decision provides the choice.

Dynamic Values

Overview

Use Dynamic Values to generate Jinja dynamically within the Playbook Designer. To make your playbook dynamic use Jinja templates to define various conditions within steps in a playbook. However, you must have some knowledge of Jinja (see Jinja Documentation) and must understand the workflow of the playbook steps in the JSON format to create Jinja templates.

Using the Jinja editor, you can apply a Jinja template to a JSON input and then render the output, thereby checking the validity of the Jinja and the output before you add the Jinja to the Playbook.

Using Dynamic Values, you can dynamically add Jinja to steps within a Playbook. Click a step, within the playbook that takes Jinja as an input and Dynamic Values is displayed. Choose from the options presented to add Jinja to the step.

FortiSOAR simplifies the process of building playbooks without requiring to have Jinja or Python knowledge. Use the "Expressions" tab on Dynamic Values to build playbooks with medium-level complexity without any programming knowledge, with the option to use Jinja or Code Snippets to build playbooks that are very complex. For more information, see Expressions Usage.

Jinja Editor

Use the Jinja editor to apply a Jinja template to a JSON input and then render the output. You can thereby check the validity of the Jinja and the output before you add the Jinja to the Playbook.

To open the Jinja Editor, in the Playbook Designer, click **Tools** > **Jinja Editor**.
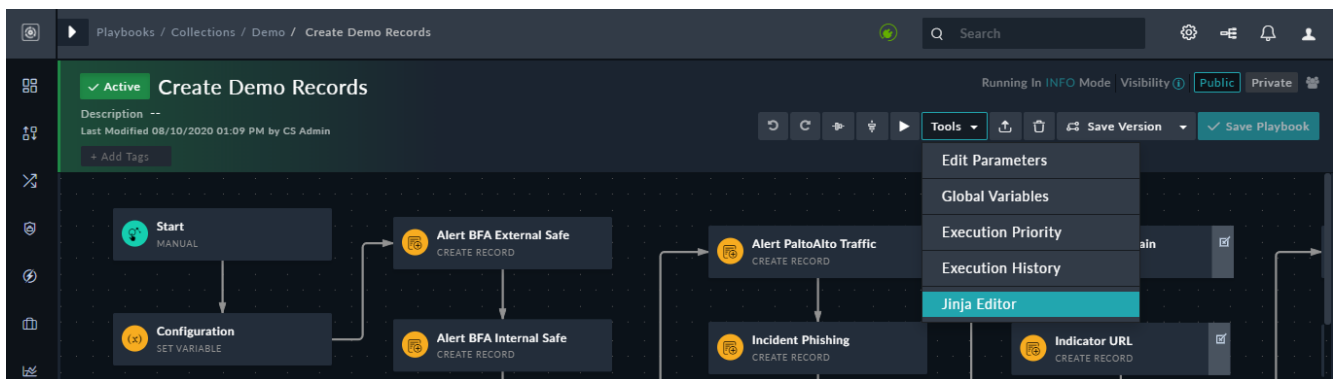
The Jinja Editor has three areas: Template, Json, and Output.

- **Jinja Template**: Use the `Jinja Template` area to specify the Jinja in *curly* brackets.
- **JSON**: Use the `JSON` area to specify the JSON input. JSON is always in the format of `"Key":"Value"` pair. If there are syntax errors in the JSON you have written, the Jinja editor displays a `Bad String` prompt. You can also specify nested key-value pairs.
  Once you have entered the Jinja and JSON, click **Render** to display the output.

- **Output**: The `Output` area displays the output that would be generated by the combination of the entered the Jinja and the JSON.
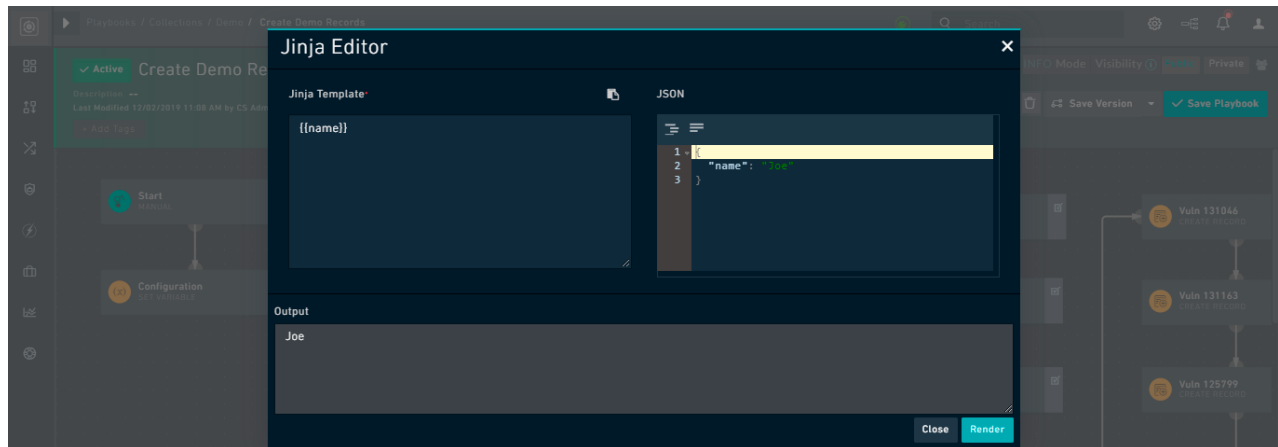
When an object is returned as the result, then the Jinja Editor will display the output as an object instead of text area.

Use the **Choose a recent playbook execution** drop-down list to select the recent execution history of the playbook and click **Load ENV JSON** to populate the JSON in the jinja editor, as shown in the following image:

Having the ability to load real JSON data from a recently executed playbook enables you to test your jinja statements in Dynamic Values.
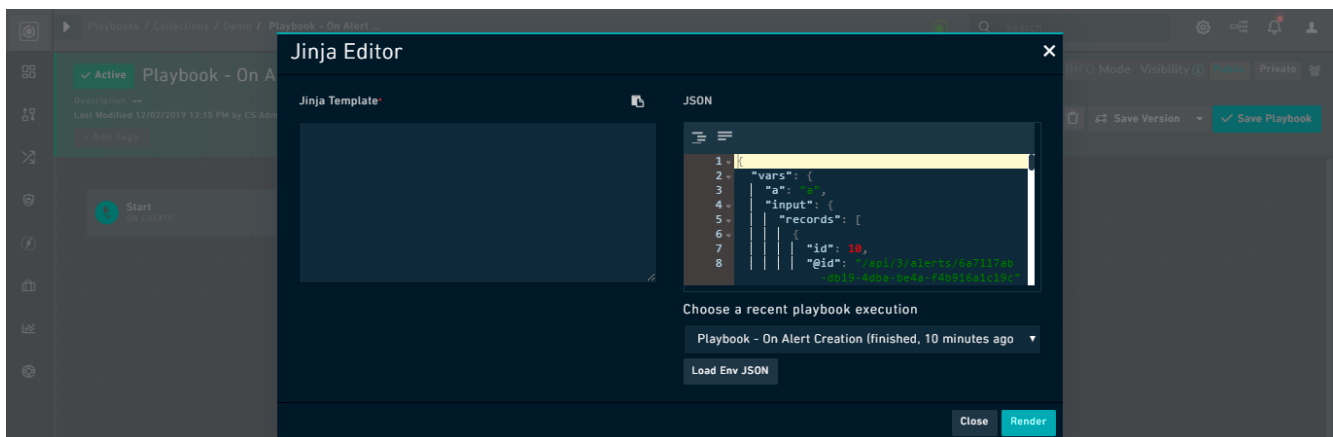
Dynamic Values Usage

Dynamic Values is used within the Playbook Designer. Use the Dynamic Values directly within steps of your playbook to dynamically add Jinja to those steps. Click a step within the playbook that takes Jinja as an input and Dynamic Values is displayed. Choose from the options presented to add Jinja to the step.

Dynamic Values is visible for input fields such as text fields, rich text, date/time fields, picklists, checkboxes, etc. You can use the **Add Custom Expression** button to toggle fields, such as drop-down lists and checkboxes, and add custom jinja expressions for fields such as picklists, Lookups, File selectors, rich text, text fields, etc. Clicking the **Add Custom Expression** button also displays the **Dynamic Values dialog**, using which also you can add expressions to these fields. The ability to add Jinja expressions to these fields enables you to customize your playbooks further.

> In version 7.0.0, FortiSOAR has updated the arrow library due to which the `timestamp` attribute has been changed into `int_timestamp` for *DateTime* jinja expressions. Therefore, new playbooks must use the `int_timestamp` for any *DateTime* jinja expressions. For more information, see the Dynamic Variables chapter.
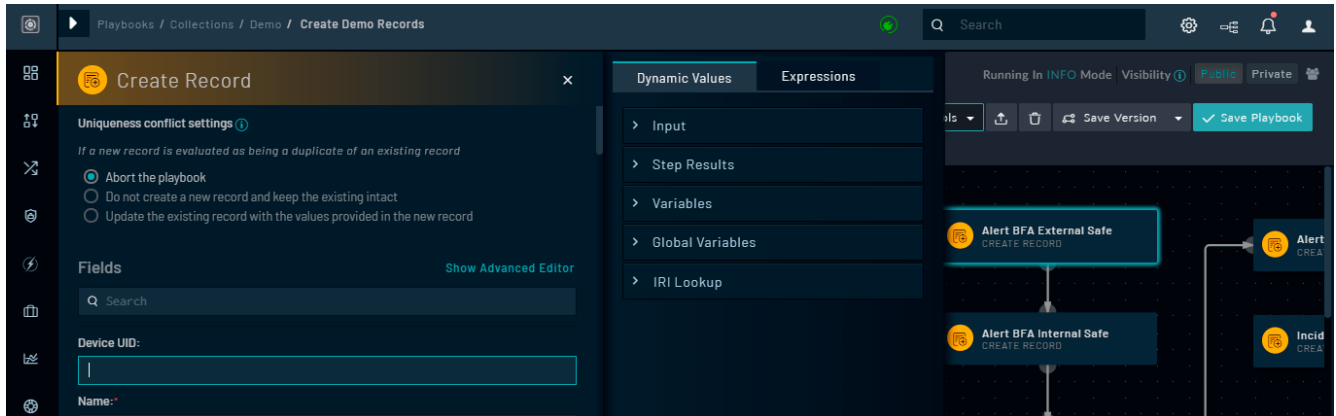
You can also use Dynamic Values within a `Text` field type that has a subtype of either `Rich Text (Markdown)`, which is the default, or `Rich Text (HTML)`. Earlier, the Dynamic Values dialog would not be displayed for a `Rich Text` type field. To display Dynamic Values within a `Rich Text` type field, click the **Dynamic Values** (![icon]) icon in the formatting toolbar as shown in the following image:
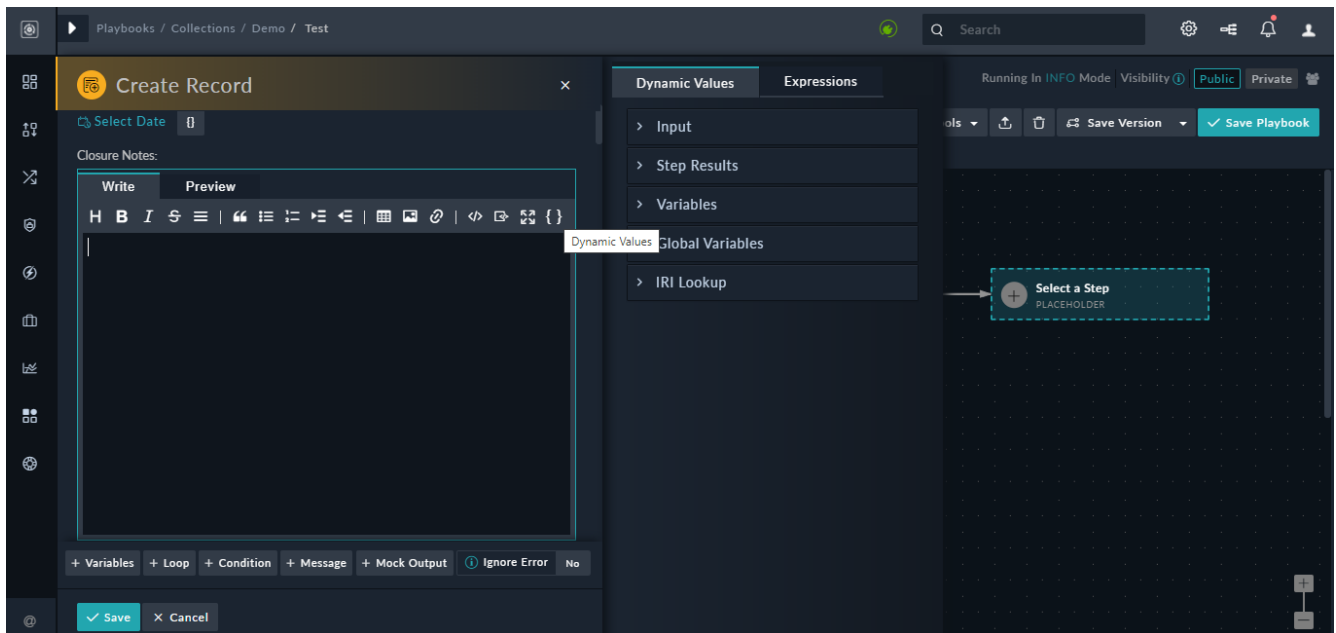
Dynamic Values provides you with the following Jinja options within a step:

- Input
- Step Results
- Variables
- Global Variables
- IRI Lookup
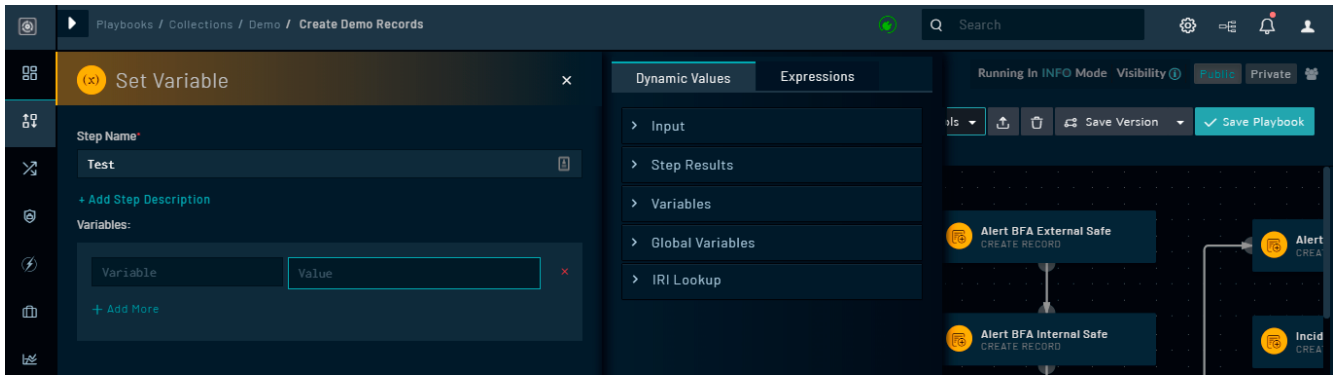
> 💡 Before you delete or modify any global variable or variable(s) ensure that you have removed or updated the variable in the Playbook to ensure that the change does not affect the functionality of the playbook.

Use the "Expressions" tab on Dynamic Values to build playbooks based on your requirements and without programming knowledge. For more information, see Expressions Usage.

Input

Use the **Input** option to add various types of data, variables, and parameters that you can use in your current step.
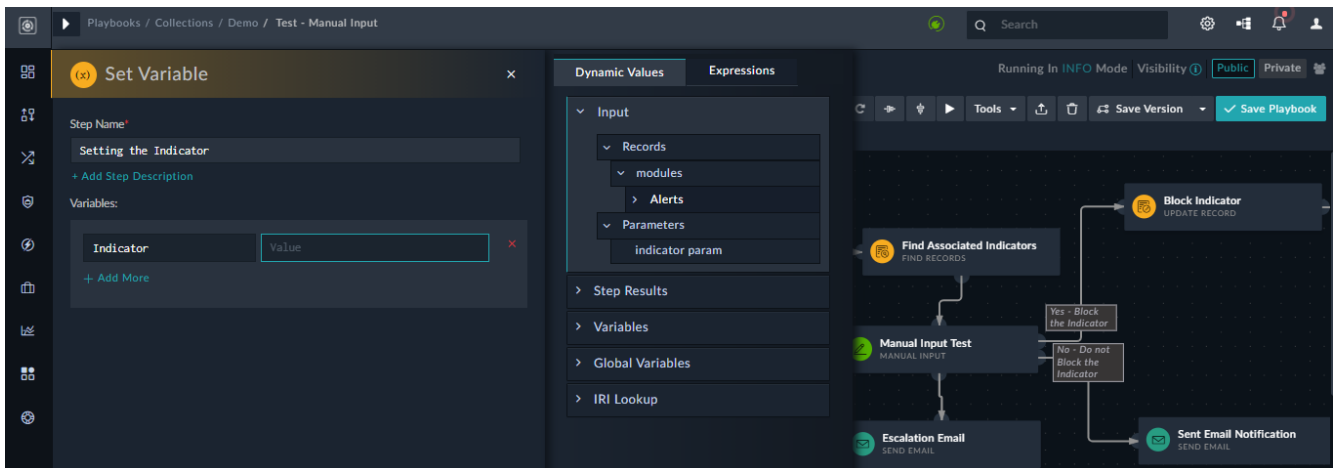
The Input option provides you various type of parameters that you can use in your current step:

- Parameters that you have defined using **Tools** > **Edit Parameters** in the playbook designer appears when you click **Input > Parameters**.
- Trigger step data has been added as part of the **Inputs** so that you can use the variables and data from the module on which trigger has been added. Data of the trigger step appears when you click **Input > Records**.
- Data of the trigger step, in case of `Custom API Endpoint Trigger`, appears when you click **Input > Parameters > api_body**.
- Variables that you have defined in the Manual Trigger step appears when you click **Input > Variables**.

> ⚠️ While importing playbooks that were created using an older version of FortiSOAR, before you use the "Input" option in any step, ensure that you open and save the trigger step and then save the playbook.

Step Results

The **Step Results** option enables you to use the output of the steps that have been executed, in the current step. Dynamic Values also displays the output of the current step in **Variables** and **Message** options in playbook steps, so that you can add or store the output of the current step directly in the step itself. For more information about playbook steps, see the Triggers & Steps chapter.

Dynamic Values displays the output of Step Results for the current step in the format `vars.result.keyname`. Dynamic Values displays the output of Step Results for the previous or already executed steps in the format `vars.steps.stepname.keyname`.

To use the output of the steps that have been executed, click the step in which you want to the use the executed steps' output, which then displays Dynamic Values. Select the **Step Results** option. Dynamic Values displays the output schema, with all its attributes, of the output of all the steps that have been executed. You can then use the output schema and attributes of any of the executed steps as an input to the current step based on the logic or functionality of the current step.

You can also use an array element in an executed steps' output:

If you select an array element for the executed steps' output, then you must specify the position of the element (index `[i]`) in the Jinja that is generated. The index value of an array starts from zero `[0]`. For example, if you want to fetch the `name` property from the `Find_all_Open_Alerts []` array from the executed steps' output, then the Jinja that is generated is as follows: `{{vars.steps.Find_all_Open_Alerts[0].name}}`.

Therefore, before you run the playbook and require to fetch any element other than the first element in the array, you must provide the position of the element. For example, if you want to fetch the `name` property of the 4th element of the

`Find_all_Open_Alerts []` array then your Jinja must be written as `{{vars.steps.Find_all_Open_Alerts [3].name}}`.

If there is no step output available or if you are at the first step in the Playbook or if the step is not connected to another step, then the Jinja generator displays the following message: `Either there is no previous step or the previous step output is not known. If there is no previous step then connect another step to the current step to view the previous step output.`

Variables

Variables are variables that can be used only in the playbook in which it is defined. Therefore, the scope of a variable is *local*. To create a Variable, in the Playbook Designer, click the **Set Variable** step and add the `Name` and `Value` for the Variable and then click **Save**.
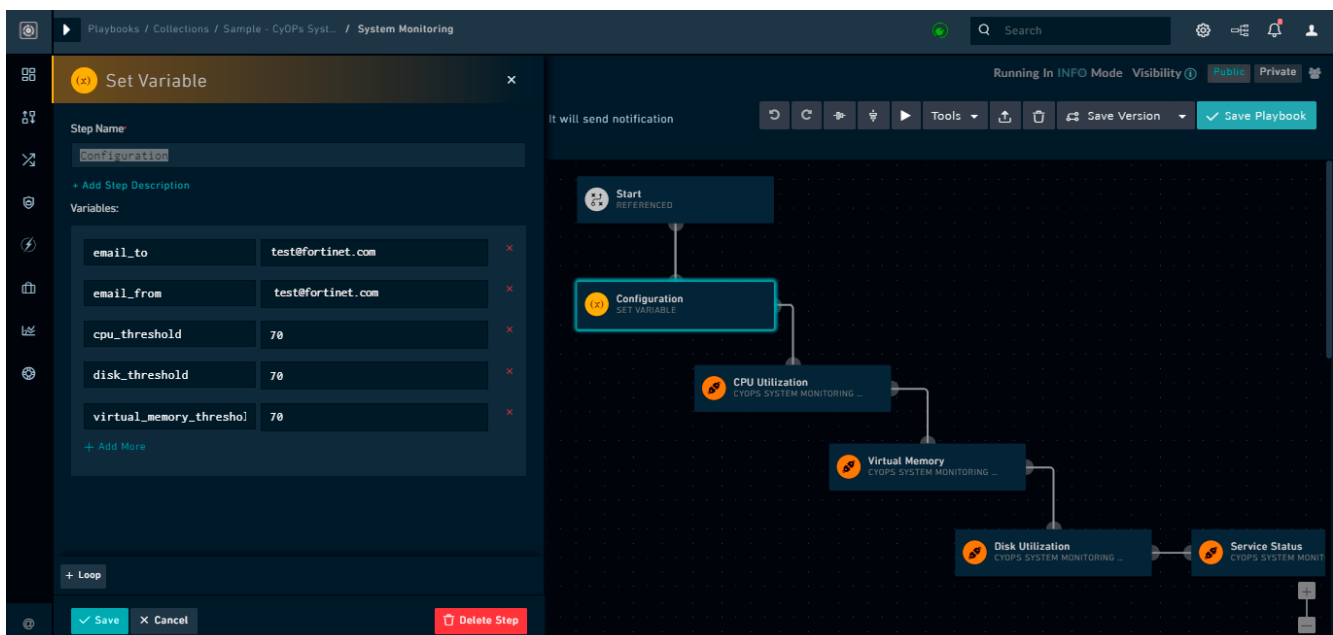
You can define multiple set variables in a playbook and then in this case you will see multiple Variables in the steps.

You use the Variable like you would use a Global Variables, except that variables can be used only in the remaining steps of the playbook in which they have been defined or in any child playbook, regardless of how many levels deep the child playbooks are called.

If there are no variables defined, the Jinja generator displays the following message: `There are no Variables declared in this playbook or the current step is not connected to any step. You can create Variables by adding 'Set Variable' step.`

Global Variables

Global variables (called macros in earlier version), are variables that can be used across playbooks. You can declare a global variable once and then use it across all playbooks, instead of having to redefine the variable every time in each playbook. You can create global variables only in the Playbook Designer. FortiSOAR includes some pre-defined global variables.

To create a global variable, in the Playbook Designer, click **Tools** > **Global Variable**. Click **New Global Variable** and enter appropriate content in the `Variable Name` and `Field Value` fields for the variable and click **Submit** to create a global variable. You can optionally also add the default value for the variable in the `Default Value` field.

> Variable Names must always begin with a character when you are creating global variables and the name can contain only alphabets and numerals. Special characters and spaces are not allowed.

To ensure that the correct hostname is displayed in links contained in emails sent by System Playbooks, you must update the `Server_fqhn` global variable in **Global Variables**. Click the `Edit` icon in the `Server_fqhn` global variable, and in the `Field Value` field add the appropriate hostname value, and then click **Submit**. If you have not specified the

hostname in global variables, then the hostname that you had specified or that was present when you installed FortiSOAR will be the default hostname and this will be added in the email. In this case, ensure that you have used the `Server_fqhn` global variable in the `Send Email` step in the playbook that is sending the email.

Example of using a global variable: In the `Set Variable` step, you need to set a name and value. When you click the `Value` field, Dynamic Values is displayed. Click **Global Variables** in Dynamic Values, and you will see a list of global variables that have been created. Click the global variable that you require, for example current date. This adds the Jinja value of the global variable in the `Value` field:
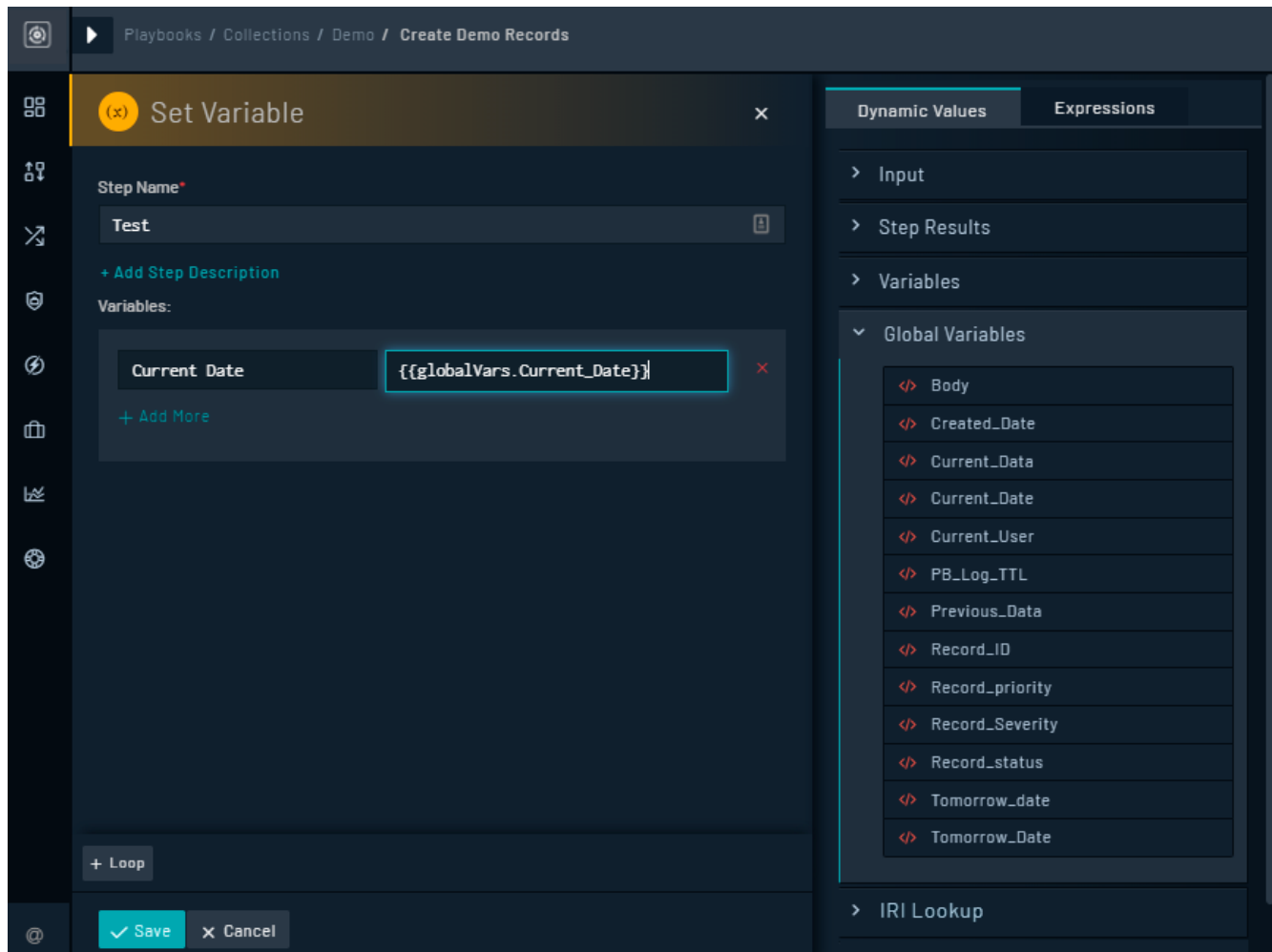
IRI Lookup

All foreign key references use International Resource Identifiers (IRIs) to reference records within the system. IRIs are generated automatically when FortiSOAR inserts records. FortiSOAR uses IRI values in multiple places for referencing picklists, playbooks, attachments, etc. Use the **IRI Lookup** option to efficiently use the IRI values of picklists, attachments, or playbooks configured in your system.

To use the IRI lookup, click the step in which you want to insert an IRI value, which then displays Dynamic Values. Select the **IRI Lookup** option and from the **Choose Entity Type** drop-down menu, choose the entity, either a **Picklist**, **Attachment**, or **Playbook**. For example, if you choose **Picklist**, then from the **Provide Entity To Lookup IRI for** drop-down menu, select the `Picklist` whose IRI value you want to add to the step. In our example, we want to add the IRI

value for the picklist that retrieves the types of alerts. Therefore, from the **Provide Entity To Lookup IRI for** drop-down menu, select **AlertType**, and then in the `Click to Add IRI Value` section, select the alert type that you want to add, for example, **Phishing**, as shown in the following image:
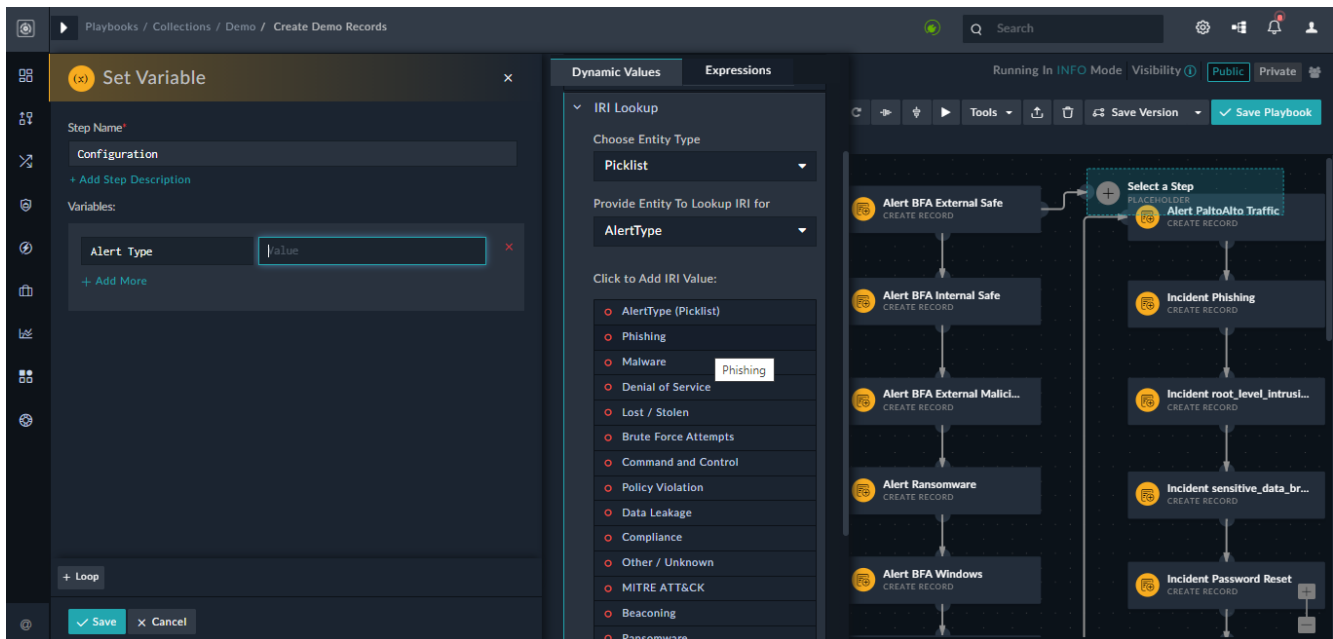

Once you click Phishing, the IRI value (jinja) of 'Phishing' alert type is added to the playbook in the Jinja format, which is `{{("picklistName"|picklist("itemvalue of picklist"))["@id"]}}`.

Expressions Usage

Use the "Expressions" tab in Dynamic Values to build playbooks based on your requirements and without programming knowledge, and add various operations and expressions to playbooks. The Expressions tab contains easy-to-understand operations that cover most aspects of playbook development. The operations are grouped logically as per their functionality, for example, if you want to convert datetime into a specific format or to a different time zone then these operations will be listed in the Date Time Operations list. Similarly, if you want to replace text in a string with a regex, then these operations will be listed in the Regex Operations list. You can search for operations using the search textbox in the Expressions tab. Each operation has an information icon that you can hover over to view more information about that particular operation.
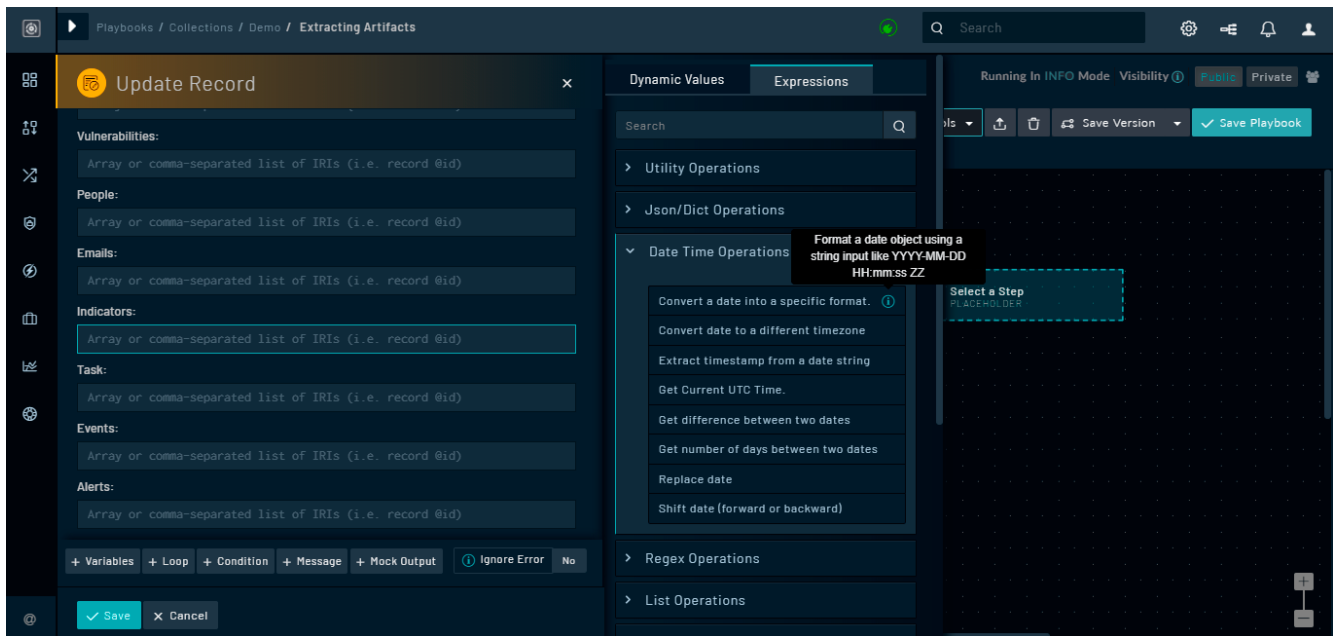
An example of using an expression would be requiring to extract artifacts from the source data of an alert that has been created in FortiSOAR from a SIEM and update that record with the extracted artifacts. You can use **Utility Operation** > **Extract artifacts from a string** operation to extract artifacts from the source data.

To use the Extract artifacts from a string operation, in the "Update Record" step, click the **Indicators** field, which displays Dynamic Values. Click the **Expressions** tab and then click the **Utility Operations** list:
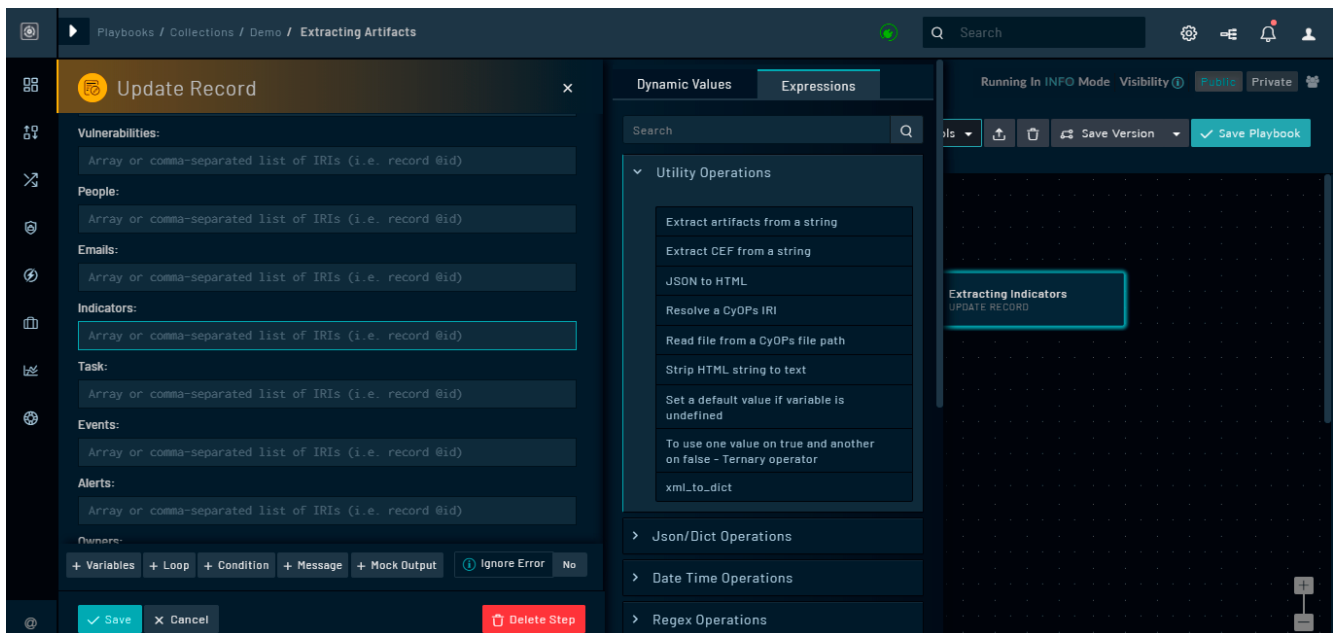
Click the **Extract artifacts from a string** operation, which in turn displays a **Inputs** text box. From the **Choose Field** list, you can select Source Data of the alert by clicking **Input > Records > modules > Alerts** and clicking the **Source Data** field:
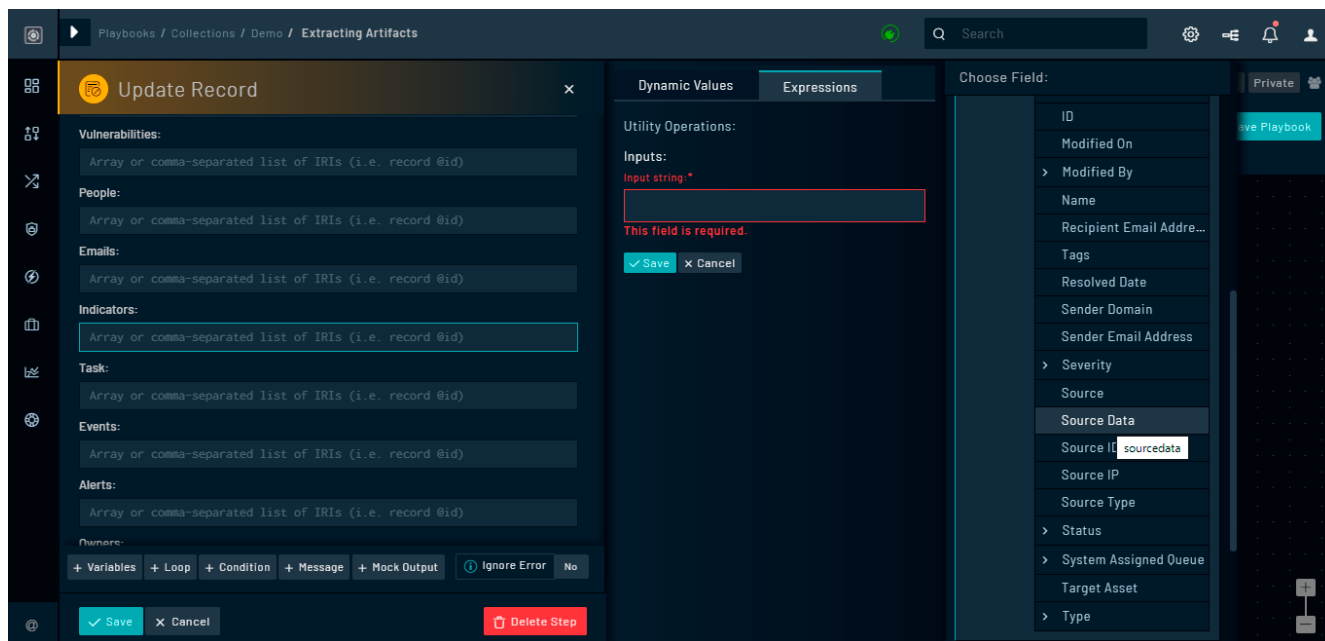
This adds the corresponding Jinja value in the **Input string** field:
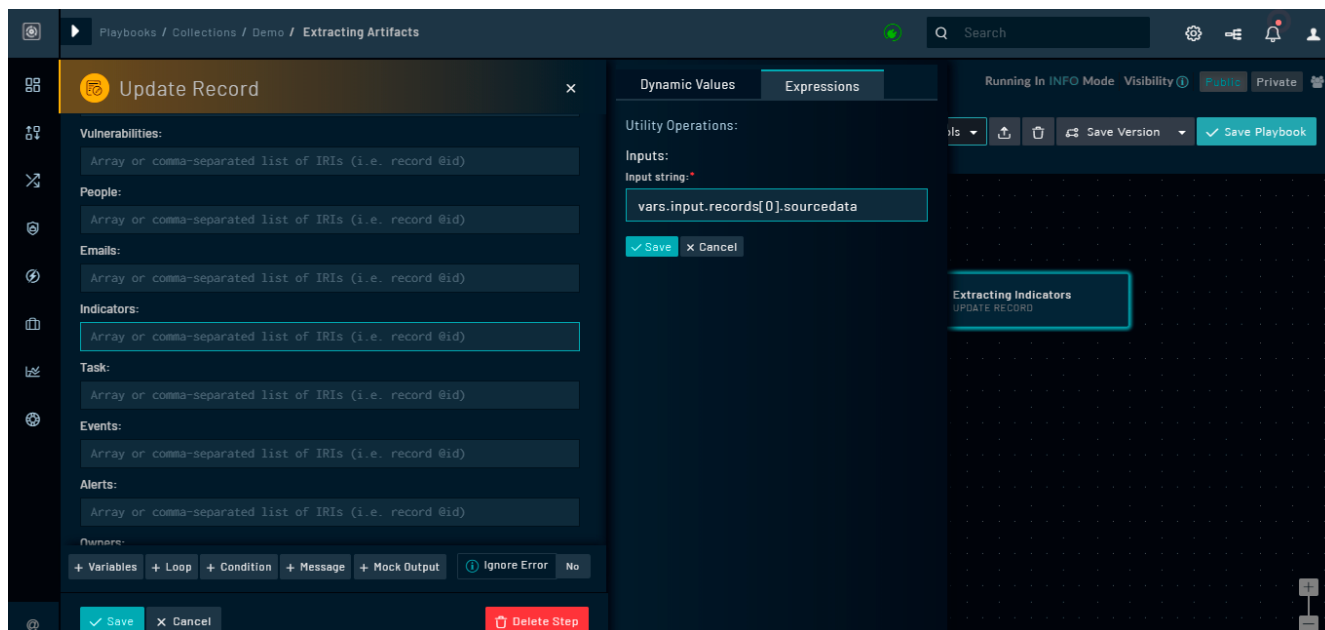
Click **Save** to add the corresponding Jinja expressions in the **Indicators** field:
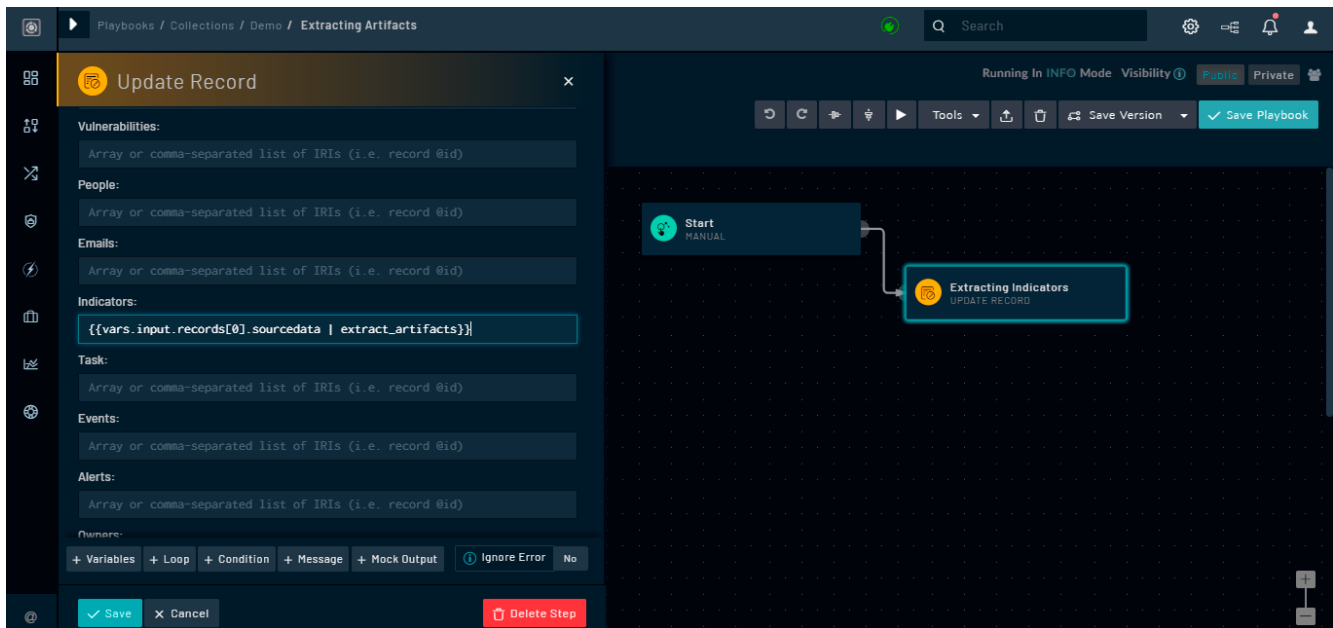
Click **Save** to save the Update Record step and then save the playbook. Now, when you run this playbook for an alert that contains source data, this step will extract artifacts from the source data and update the indicators associated with the alert record with the extracted artifacts.

Whenever FortiSOAR is upgraded, the files located in the `/opt/cyops-workflow/sealab/expression_builder/expressions` folder will be overridden based on enhancements or additions made to the expressions. Therefore, you should make changes to the expressions in the so it is advised to the user that they should make the changes to expression in the files located in the `/opt/cyops-workflow/sealab/expression_builder/custom` folder.

Adding your own expressions

You can also create your own expressions and add it to the "Expressions" tab in Dynamic Values. You can either use existing Jinja2 filters (https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html) or create your own new function and add it to the "Expressions" tab in Dynamic Values.

Adding existing Jinja2 filters to the "Expressions" tab

You can add existing Jinja2 filters that are currently not part of "Expressions" tab. An example of this can be the `{{ path | win_splitdrive }}` filter, which separates the Windows drive letter from the rest of a file path. To add this filter to the "Expressions" tab, do the following:

1. SSH to your FortiSOAR VM and login as a *root* user.
2. Navigate to the `/opt/cyops-workflow/sealab/expression_builder/custom` folder.
3. Edit the `my_expressions.json` file and add the `{{ path | win_splitdrive }}` filter as follows:

   ```
   {
     "name": "my_expressions",
   ```

```
      "title": "My Expressions",
      "description": "",
      "operations": [
        {
          "name": "Seperate Windows Drive Letter",
          "jexp": "{{params.path}} | win_splitdrive",
          "description": "",
          "params": [
            {
              "name": "path",
              "title": "File Path",
              "type": "text",
              "value": "",
              "editable": true,
              "required": true
            }
          ]
        }
      ]
    }
```

4. Save the `my_expressions.json` file.
   Note: The `jexp` parameter displays the final expression that will be added to the field in the playbook.
5. Restart the `uwsgi` and `celeryd` services using the following commands:
   ```
   # systemctl restart uwsgi
   # systemctl restart celeryd
   ```

Once you have restarted the `uwsgi` and `celeryd` services, logon to FortiSOAR and open a playbook to view the "Expressions" tab in Dynamic Values. You will see "My Expressions" added at the end of the "Expressions" list:
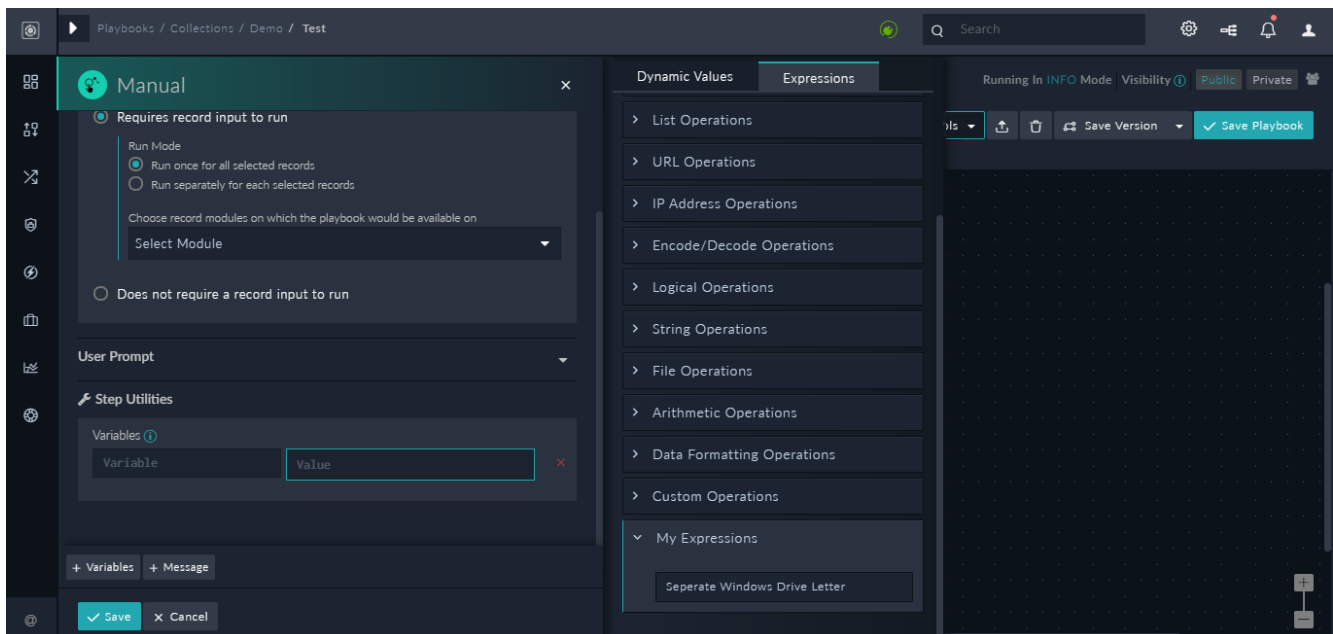
Click **Separate Windows Drive Letter** to display the inputs to be gathered from the user and enter the file path from which you want to separate the windows drive letter:
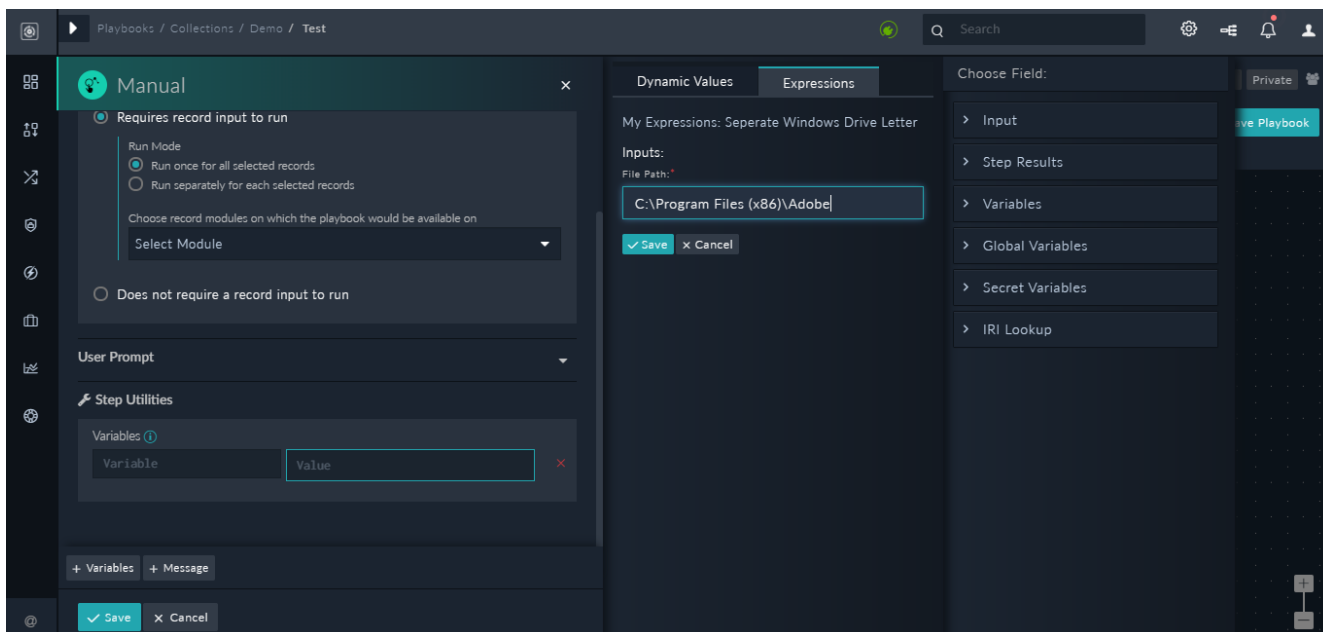
Click **Save** to display the final expression that will be added to the field in the playbook:
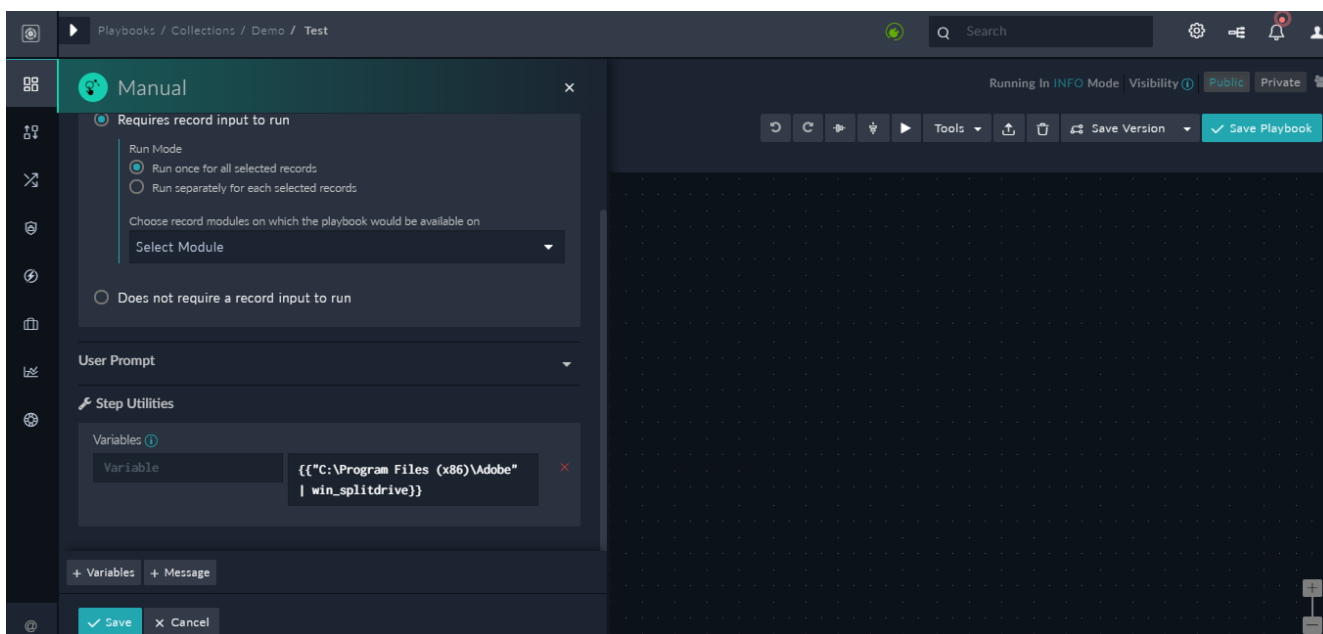
Creating a new function and then adding it to the "Expressions" tab

You can also create a new function and either use it as a filter or add it to the "Expressions" tab in Dynamic Values.

To add a new function that displays a sum of two numbers, do the following:

1. SSH to your FortiSOAR VM and login as a *root* user.
2. Navigate to the `/opt/cyops-workflow/sealab/expression_builder/custom` folder.

3. Edit the `custom_functions.py` file as follows:

    ```
    vi custom_functions.py

    def sum(a,b):
         return (a+b)

    func_map = {
        'sum': sum
    }
    ```

4. Save the `custom_functions.py` file.

5. Restart the `uwsgi` and `celeryd` services using the following commands:

    ```
    # systemctl restart uwsgi
    # systemctl restart celeryd
    ```

6. Now, you can use the `sum` filter as required or you can add it to the "Expressions" tab in Dynamic Values, using the method described in the earlier section.

Dynamic Variables

Overview

Dynamic variables are objects that can be set and accessed within a playbook. Any valid Python object can be a dynamic variable. This includes ints, strings, dictionaries, etc. Variables themselves have no type information associated with them; however, playbook steps do. Steps may attempt to coerce dynamic variables into the expected data type; however, it is mostly on the caller to pass the correct types.

Dynamic variables can be passed to playbook steps as arguments directly, or they may be embedded in a larger string, where they will act more as global variables (or macros), getting replaced by a string representation of themselves.

Syntax

Double curly braces ({{ }}) demarcate dynamic variables from the surrounding text. Anything that goes between the braces is a dynamic variable. The most basic use of the dynamic variable is as a simple dictionary lookup.

> The general data structure you are using matters within the usage of the dynamic variable. JSON is the easiest data format to consume and use. XML may be converted into JSON directly so that it may also be used. The following examples assume that you are able to use a JSON format.

Let's look at some examples. Say you have an object (array) named `users` which has the following structure:

```
{
    'Alvian': 42,
    'Kreb': 413,
    'Mandu': 1
}
```

Example 1

You can then use dynamic variables to access the values of that object.

```
{{ vars.users.Alvian }}
```

This statement will evaluate to 42.

Example 2

```
There are {{ vars.users.Kreb }} Krebs in FortiSOAR.
```

This statement will evaluate to the string "There are 413 Krebs in FortiSOAR."

Example 3

```
{{ vars.users.does_not_exist }}
```

This statement would evaluate to an error and would be displayed as:

```
no such element: users['does_not_exist']
```

Example 4

Say you modified the object (array) named `users` to have the following structure:

```
{
    'Alvian': 42,
    'Kreb': {
        'original': 413,
        'pi': 3.14
    },
    'Mandu': 1
}
```

To access the secondary array is as easy as adding an additional key for the key-value pair you desire to access.

```
{{ vars.users.Krebs.pi }}
```

This statement would evaluate to 3.14. An alternative format for accessing the variable, which may be used in case of special characters present, is:

```
{{ vars.users['Krebs']['pi'] }}
```

This statement would evaluate identically to the previous, 3.14.

Implementation

The major driving force behind dynamic variables is Jinja2 templates. A general overview of how Jinja2 works can be found here.

Specifically, to render a template, Jinja takes two arguments: a context and a template string. The template string is the dynamic variable itself, which is provided by users in the playbook. The context object, on the other hand, will be created automatically before each and every playbook step. It contains various helper functions as well as the internal representation of the dynamic variable data.

Scope

Scope for dynamic variables is defined by the `COPY_ENV_FOR_REFERENCE_WORKFLOW` setting. Use this setting to pass variables to a reference playbook.

By default, the `COPY_ENV_FOR_REFERENCE_WORKFLOW` is set to `false`.

Functionality

There are several top-level objects that can be accessed within a dynamic variable.

Dictionary-like Objects

Most ordinary variables are stored under the vars namespace. Whenever a variable is declared using: class:`workflow.tasks.set_variable`, it will go under `vars`. Additionally, the playbook engine will automatically set the following variables:

- `vars.result`: This contains the return value of the previous playbook step.
- `vars.input.records`: This contains information about what triggered a playbook, i.e., the body of the inbound request.
- `vars.request.headers`: This contains the metadata of all the headers that are part of the playbooks environment, and which can be used in the playbooks, such as `X-RUNBYUSER` which is a jinja template to retrieve the name of the user who triggered the playbook. Some other parameters are, trigger type, authorization, accept, host, content-type, etc.
- `vars.input.params.api_body`: This contains the data passed from a Custom API Endpoint trigger.

Built-in Functions & Filters

Functions

- `arrow`: Datetime functions:

```
{{ arrow.utcnow().int_timestamp }}
```

In version 7.0.0, FortiSOAR has updated the arrow library, due to which the `timestamp` attribute has been changed into `int_timestamp` for *DateTime* jinja expressions, . For more information see, https://arrow.readthedocs.io/en/latest/releases.html#id4

> New playbooks must use the `int_timestamp` for any *DateTime* jinja expressions.

More documentation can be found here

- `uuid`: returns a uuid using python's `uuid.uuid4()` function

```
{{ uuid() }}
```

Filters

See the Jinja Filters and Functions chapter for information.

FAQS

How are dynamic variables used in condition steps?

Decision steps use dynamic variables with logical equalities of the form:

```
{{ 8 == 8 }}
```

This statement will return either the string 'True,' or 'False' which will automatically be converted into a real boolean value.

Decision steps advanced interface does not require the use of curly braces like `{{ }}`.

Jinja Filters and Functions

Overview

Use jinja2 filters to design and manipulate playbook step outputs. Jinja operations are supported in the Playbook Engine and you can also use the Custom Functions and Filters that are documented in this chapter.

> All filters are case-sensitive.

These examples present in this chapter provide a reference to common and very useful string operations that may be leveraged within the Playbook engine.

Filters

FortiSOAR supports the following filters:

- `fromIRI`: Will resolve an IRI and return the object(s) that live(s) there. This is similar to loading the object by id (IRI).
  `{{ '/api/3/events/8' | fromIRI }}{{ vars.event.alert_iri | fromIRI }}`
  You can use dot access for values returned by fromIRI.
  For example: To get a person record and return their 'name' field you can use the following:
  `{{ (vars.person_iri | fromIRI).name }}`
  You can also use fromIRI recursively, for example:
  `{{ ((vars.event.alert |fromIRI).owner| fromIRI).name }}`
  You can also retrieve relationship data for a record on which a playbook is running, for example:
  `{{('/api/3/alerts/<alert_IRI>?$relationships=true' | fromIRI).indicators}}`
- `toDict`: attempt to coerce a string into a dictionary for access.
  `{{ (request.data.incident_string | toDict).id }}`
- `xml_to_dict`: Converts an XML string into a dictionary for access:
  `{{ '<?xml version="1.0" ?><person><name>john</name><age>20</age></person>' | xml_to_dict }}`
- `extract_artifacts`: Parses and extracts a list of IOCs from a given string:
  `{{'abc.com 192.168.42.23' | extract_artifacts}}`
- `parse_cef`: Parses a given CEF string and converts the CEF string into a dictionary:
  `{{ 'some string containing cef' | parse_cef }}`
- `readfile`: Fetches the contents of a file that is downloaded in FortiSOAR:
  `{{ vars.result | readfile}}`
  where `vars.result` is the name of the file.
- `ip_range`: Checks if the IP address is in the specified range:
  {{vars.ip | ip_range('198.162.0.0/24')}}
- `counter`: Gets the count of each item's occurrence in an array of items:
  `{{data| counter}}`

For example:
```
data: ['apple','red','apple','red','red','pear']
{{data| counter}}
```
Result: {'red': 3, 'apple': 2, 'pear': 1}

FortiSOAR also supports following filters, more information for which is present at
http://docs.ansible.com/ansible/latest/playbooks_filters.html.

Filters for formatting data

The following filters take a data structure in a template and render it in a slightly different format. These are occasionally useful for debugging:

```
{{ some_variable | to_json }}
{{ some_variable | to_yaml }}
```

For human readable output, you can use:

```
{{ some_variable | to_nice_json }}
{{ some_variable | to_nice_yaml }}
```

It is also possible to change the indentation the variables:

```
{{ some_variable | to_nice_json(indent=2) }}
{{ some_variable | to_nice_yaml(indent=8) }}
```

Alternatively, you may be reading in some already formatted data:

```
{{ some_variable | from_json }}
{{ some_variable | from_yaml }}
```

Filters that operate on list variables

To get the minimum value from a list of numbers:

```
{{ list1 | min }}
```

To get the maximum value from a list of numbers:

```
{{ [3, 4, 2] | max }}
```

Filters that return a unique set from sets or lists

To get a unique set from a list:

```
{{ list1 | unique }}
```

To get a union of two lists:

```
{{ list1 | union(list2) }}
```

To get the intersection of 2 lists (unique list of all items in both):

```
{{ list1 | intersect(list2) }}
```

To get the difference of 2 lists (items in 1 that don't exist in 2):

```
{{ list1 | difference(list2) }}
```

To get the symmetric difference of 2 lists (items exclusive to each list):

```
{{ list1 | symmetric_difference(list2) }}
```

Random Number filter

The following filters can be used similar to the default jinja2 random filter (returning a random item from a sequence of items), but they can also be used to generate a random number based on a range.

To get a random item from a list:

```
{{ ['a','b','c']|random }}
# => c
```

To get a random number from 0 to supplied end:

```
{{ 59 |random}}
# => 21
```

Get a random number from 0 to 100 but in steps of 10:

```
{{ 100 |random(step=10) }}
# => 70
```

Get a random number from 1 to 100 but in steps of 10:

```
{{ 100 |random(1, 10) }}
# => 31
{{ 100 |random(start=1, step=10) }}
# => 51
```

To initialize the random number generator from a seed. This way, you can create random-but-idempotent numbers:

```
{{ 59 |random(seed=inventory_hostname) }}
```

Shuffle filter

The following filters randomize an existing list, giving a different order every invocation.

To get a random list from an existing list:

```
{{ ['a','b','c']|shuffle }}
# => ['c','a','b']

{{ ['a','b','c']|shuffle }}
# => ['b','c','a']
```

To shuffle a list idempotent. For this you will require a seed:

```
{{ ['a','b','c']|shuffle(seed=inventory_hostname) }}
# => ['b','a','c']
```

When this filter is used with a non 'listable' item it is a noop. Otherwise, it always returns a list.

Filters for math operations

To get the logarithm (default is e):

```
{{ myvar | log }}
```

To get the base 10 logarithm:

```
{{ myvar | log(10) }}
```

To get the power of 2! (or 5):

```
{{ myvar | pow(2) }}
{{ myvar | pow(5) }}
```

To get the square root, or the 5th:

```
{{ myvar | root }}
{{ myvar | root(5) }}
```

IP Address filters

To test if a string is a valid IP address:

```
{{ myvar | ipaddr }}
```

To get the IP address in a specific IP protocol version:

```
{{ myvar | ipv4 }}
{{ myvar | ipv6 }}
```

To extract specific information from an IP address. For example, to get the IP address itself from a CIDR, you can use:

```
{{ '192.0.2.1/24' | ipaddr('address') }}
```

To filter a list of IP addresses:

```
test_list = ['192.24.2.1', 'host.fqdn', '::1', '192.168.32.0/24', 'fe80::100/10', True,
'', '42540766412265424405338506004571095040/64']
```

```
# {{ test_list | ipaddr }}
['192.24.2.1', '::1', '192.168.32.0/24', 'fe80::100/10', '2001:db8:32c:faad::/64']
```

```
# {{ test_list | ipv4 }}
['192.24.2.1', '192.168.32.0/24']
```

```
# {{ test_list | ipv6 }}
['::1', 'fe80::100/10', '2001:db8:32c:faad::/64']
```

To get a host IP address from a list of IP addresses:

```
# {{ test_list | ipaddr('host') }}
['192.24.2.1/32', '::1/128', 'fe80::100/10']
```

To get a public IP address from a list of IP addresses:

```
# {{ test_list | ipaddr('public') }}
['192.24.2.1', '2001:db8:32c:faad::/64']
```

To get a private IP address from a list of IP addresses:

```
# {{ test_list | ipaddr('private') }}
['192.168.32.0/24', 'fe80::100/10']
```

Network range as a query:

```
# {{ test_list | ipaddr('192.0.0.0/8') }}
['192.24.2.1', '192.168.32.0/24']
```

Hashing filters

To get the sha1 hash of a string:

```
{{ 'test1'|hash('sha1') }}
```

To get the md5 hash of a string:

```
{{ 'test1'|hash('md5') }}
```

To get a string checksum:

```
{{ 'test2'|checksum }}
```

Other hashes (platform dependent):

```
{{ 'test2'|hash('blowfish') }}
```

To get a sha512 password hash (random salt):

```
{{ 'passwordsaresecret'|password_hash('sha512') }}
```

To get a sha256 password hash with a specific salt:

```
{{ 'secretpassword'|password_hash('sha256', 'mysecretsalt') }}
```

FortiSOAR uses the `haslib` library for `hash` and `passlib` library for `password_hash`.

Filters for combining hashes and dictionaries

The `combine` filter allows hashes to be merged. For example, the following would override keys in one hash:

```
{{ {'a':1, 'b':2}|combine({'b':3}) }}
```

The resulting hash would be:

```
{'a':1, 'b':3}
```

The filter also accepts an optional `recursive=True` parameter to not only override keys in the first hash, but also recursively into nested hashes and merge their keys too:

```
{{ {'a':{'foo':1, 'bar':2}, 'b':2}|combine({'a':{'bar':3, 'baz':4}}, recursive=True) }}
```

The resulting hash would be:

```
{'a':{'foo':1, 'bar':3, 'baz':4}, 'b':2}
```

The filter can also take multiple arguments to merge:

```
{{ a|combine(b, c, d) }}
```

In this case, keys in d would override those in c, which would override those in b, and so on.

Filters for extracting values from containers

The extract filter is used to map from a list of indices to a list of values from a container (hash or array):

```
{{ [0,2] |map('extract', ['x','y','z'])|list }}
{{ ['x','y'] |map('extract', {'x': 42, 'y': 31})|list }}
```

The results of the above expressions would be:

```
['x', 'z']
[42, 31]
```

The filter can take another argument:

```
{{ groups['x'] |map('extract', hostvars, 'ec2_ip_address')|list }}
```

This takes the list of hosts in group 'x,' looks them up in hostvars, and then looks up the ec2_ip_address of the result. The final result is a list of IP addresses for the hosts in group 'x.'

The third argument to the filter can also be a list, for a recursive lookup inside the container:

```
{{ ['a'] |map('extract', b, ['x','y'])|list }}
```

This would return a list containing the value of b['a']['x']['y'].

Comment filter

The comment filter allows you to decorate the text with a chosen comment style. For example, the following

```
{{ "Plain style (default)" | comment }}
```

will produce the following output:

```
#
# Plain style (default)
#
```

Similarly you can apply style to the comments for C (//...), C block (/*...*/), Erlang (%...) and XML (<!--...-->):

```
{{ "C style" | comment('c') }}
{{ "C block style" | comment('cblock') }}
{{ "Erlang style" | comment('erlang') }}
{{ "XML style" | comment('xml') }}
```

It is also possible to fully customize the comment style:

```
{{ "Custom style" | comment('plain', prefix='#######\n#', postfix='#\n#######\n ###\n
#') }}
```

which creates the following output:

```
#######
#
# Custom style
#
#######
   ###
    #
```

URL Split filter

The `urlsplit` filter extracts the fragment, hostname, netloc, password, path, port, query, scheme, and username from an URL. If you do not provide any arguments to this filter then it returns a dictionary of all the fields:

```
{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('hostname') }}
# => 'www.acme.com'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('netloc') }}
# => 'user:password@www.acme.com:9000'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('username') }}
# => 'user'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('path') }}
# => '/dir/index.html'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('port') }}
# => '9000'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('scheme') }}
# => 'http'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit('query') }}
# => 'query=term'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" |
urlsplit }}
# =>
# {
# "fragment": "fragment",
# "hostname": "www.acme.com",
# "netloc": "user:password@www.acme.com:9000",
# "password": "password",
# "path": "/dir/index.html",
# "port": 9000,
# "query": "query=term",
# "scheme": "http",
# "username": "user"
# }
```

Regular Expression filters

To search a string with a regex, use the `regex_search` filter:

```
# search for "foo" in "foobar"
{{ 'foobar' | regex_search('(foo)') }}

# will return empty if it cannot find a match
{{ 'ansible' | regex_search('(foobar)') }}
```

To search for all occurrences of regex matches, use the `regex_findall` filter:

```
# Return a list of all IPv4 addresses in the string
{{ 'Some DNS servers are 8.8.8.8 and 8.8.4.4' | regex_findall('\b(?:[0-9]{1,3}\.){3}[0-
9]{1,3}\b') }}
```

To replace text in a string with regex, use the `regex_replace` filter:

```
# convert "ansible" to "able"
{{ 'ansible' | regex_replace('^a.*i(.*)$', 'a\\1') }}

# convert "foobar" to "bar"
{{ 'foobar' | regex_replace('^f.*o(.*)$', '\\1') }}

# convert "localhost:80" to "localhost, 80" using named groups
{{ 'localhost:80' | regex_replace('^(?P<host>.+):(?P<port>\\d+)$', '\\g<host>,
\\g<port>') }}

# convert "localhost:80" to "localhost"
{{ 'localhost:80' | regex_replace(':80') }}
```

To escape special characters within a regex, use the `regex_escape` filter:

```
# convert '^f.*o(.*)$' to '\^f\.\*o\(\.\*\)\$'
{{ '^f.*o(.*)$' | regex_escape() }}
```

Other useful filters

To add quotes for shell usage:

```
{{ string_value | quote }}
```

To use one value on true and another on false:

```
{{ (name == "John") | ternary('Mr','Ms') }}
```

To concatenate a list into a string:

```
{{ list | join(" ") }}
```

To get the last name of a file path, like `foo.txt` out of `/etc/asdf/foo.txt`:

```
{{ path | basename }}
```

To get the last name of a windows style file path:

```
{{ path | win_basename }}
```

To separate the windows drive letter from the rest of a file path:

```
{{ path | win_splitdrive }}
```

To get only the windows drive letter:

```
{{ path  |win_splitdrive| first }}
```

To get the rest of the path without the drive letter:

```
{{ path  |win_splitdrive| last }}
```

To get the directory from a path:

```
{{ path | dirname }}
```

To get the directory from a windows path:

```
{{ path | win_dirname }}
```

To expand a path containing a tilde (~) character:

```
{{ path | expanduser }}
```

To get the real path of a link:

```
{{ path | realpath }}
```

To get the relative path of a link, from a start point:

```
{{ path | relpath('/etc') }}
```

To get the root and extension of a path or filename:

```
# with path == 'nginx.conf' the return would be ('nginx', '.conf')
{{ path | splitext }}
```

To work with Base64 encoded strings:

```
{{ encoded | b64decode }}
{{ decoded | b64encode }}
```

To create a UUID from a string:

```
{{ hostname | to_uuid }}
```

To get date object from string use the `to_datetime` filter:

```
# get amount of seconds between two dates, default date format is %Y-%m-%d %H:%M:%S
but you can pass your own one
{{ (("2016-08-14 20:00:12" |to_datetime) - ("2015-12-25"|to_datetime('%Y-%m-%d'))).seconds
}}
```

Combination filters

This set of filters returns a list of combined lists.

To get permutations of a list:

To get the largest permutations (order matters):

```
{{ [1,2,3,4,5] |permutations|list }}
```

To get the permutations of sets of three:

```
{{ [1,2,3,4,5] |permutations(3)|list }}
```

Combinations always require a set size:

To get the combinations for sets of two:

```
{{ [1,2,3,4,5] |combinations(2)|list }}
```

To get a list combining the elements of other lists use `zip`:

To get a combination of two lists:

```
{{ [1,2,3,4,5] |zip(['a','b','c','d','e','f'])|list }}
```

To get the shortest combination of two lists:

```
{{ [1,2,3] |zip(['a','b','c','d','e','f'])|list }}
```

To always exhaust all lists use `zip_longest`:

To get the longest combination of all three lists, fill with X:

```
{{ [1,2,3] |zip_longest(['a','b','c','d','e','f'], [21, 22, 23], fillvalue='X')|list }}
```

To format a date using a string (like with the shell date command), use the `strftime` filter:

```
# Display year-month-day
{{ '%Y-%m-%d' | strftime }}

# Display hour:min:sec
{{ '%H:%M:%S' | strftime }}

# Use ansible_date_time.epoch fact
{{ '%Y-%m-%d %H:%M:%S' | strftime(ansible_date_time.epoch) }}

# Use arbitrary epoch value
{{ '%Y-%m-%d' | strftime(0) }}          # => 1970-01-01
{{ '%Y-%m-%d' | strftime(1441357287) }} # => 2015-09-04
```

Debugging filters

Use the '`type_debug`' filter to display the underlying Python type of a variable. This can be useful in debugging in cases where you might need to know the exact type of a variable:

```
{{ myvar | type_debug }}
```

FortiSOAR also supports following built-in filters from Jinja, more information for which is present at Template Designer Documentation — Jinja Documentation (2.11.x).

- `abs` (*number*): Returns the absolute value of the argument.
- `attr` (*obj*, *name*): Gets an attribute of an object. `foo|attr("bar")` works like `foo.bar` just that always an attribute is returned and items are not looked up.
  See Notes on subscriptions for more details.
- `batch` (*value*, *linecount*, *fill_with=None*): Batches items. It works pretty much like slice just the other way around. It returns a list of lists with the given number of items. If you provide a second parameter, this is used to fill up missing items. For example:

```
{% for row in items|batch(3, 'FillerString') %}
    {% for column in row %}
        {{ column }}
```

```
    {% endfor %}
  {% endfor %}
```

- `capitalize`(*s*): Capitalizes a value. The first character will be uppercase, all others lowercase.
- `center`(*value, width=80*): Centers the value in a field of a given width.
- `default`(*value, default_value=u", boolean=False*): If the value is undefined it will return the passed default value, otherwise the value of the variable:

```
{{ my_variable|default('my_variable is not defined') }}
```

This would output the value of `my_variable` if the variable was defined. Otherwise, `my_variable is not defined`. If you want to use default with variables that evaluate to false, you have to set the second parameter to true:

```
{{ ''|default('the string was empty', true) }}
```

- `dictsort`(*value, case_sensitive=False, by='key'*): Sorts a dict and yields (key, value) pairs. Because python dicts are unsorted you might want to use this function to order them by either key or value:

```
{% for item in mydict|dictsort %}
    sort the dict by key, case insensitive

{% for item in mydict|dictsort(true) %}
    sort the dict by key, case sensitive

{% for item in mydict|dictsort(false, 'value') %}
    sort the dict by value, case insensitive
```

- `escape`(*s*): Converts the characters &, <, >, ', and " in strings to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.
- `filesizeformat`(*value, binary=False*): Formats the value like a human-readable file size (i.e. 13 kB, 4.1 MB, 102 Bytes, etc). Per default decimal prefixes are used (Mega, Giga, etc.) if the second parameter is set to `True` the binary prefixes are used (Mebi, Gibi).-
- `first`(*seq*): Returns the first item of a sequence.
- `float`(*value, default=0.0*): Converts the value into a floating point number. If the conversion doesn't work, it will return `0.0`. You can override this default using the first parameter.
- `forceescape`(*value*): Enforces HTML escaping. This will probably double escape variables.
- `format`(*value, args,**kwargs*): Applies python string formatting on an object:

```
{{ %s"|format("Hello?", "Foo!") }}
    -> Hello? - Foo!
```

- `groupby`(*value, attribute*): Groups a sequence of objects by a common attribute. If you, for example, have a list of dicts or objects that represent persons with gender, first_name, and last_name attributes and you want to group all users by genders you can do something like the following snippet:

```
{% for group in persons|groupby('gender') %}
      {{ group.grouper }}
    {% for person in group.list %}
      {{ person.first_name }} {{ person.last_name }}
    {% endfor %}
{% endfor %}
```

Additionally, it is possible to use tuple unpacking for the grouper and list:

```
{% for grouper, list in persons|groupby('gender') %}
    ...
{% endfor %}
```

As you can see the item, we are grouping by are stored in the grouper attribute, and the list contains all the objects that have this grouper in common. You can also use dotted notation to group by the child attribute of another attribute.

- `indent`(*s, width=4, indentfirst=False*): Returns a copy of the passed string, each line indented by four spaces. The first line is not indented. If you want to change the number of spaces or indent the first line too you can pass additional parameters to the filter:

    ```
    {{ mytext|indent(2, true) }}
        indent by two spaces and indent the first line too.
    ```

- `int`(*value, default=0, base=10*): Converts the value into an integer. If the conversion does not work, it will return 0. You can override this default using the first parameter. You can also override the default base (10) in the second parameter, which handles input with prefixes such as 0b, 0o and 0x for bases 2, 8 and 16 respectively. The base is ignored for decimal numbers and non-string values.

- `join`(*value, d=u'', attribute=None*): Returns a string which is the concatenation of the strings in the sequence. The separator between elements is an empty string per default; you can define it with the optional parameter:

    ```
    {{ [1, 2, 3] |join('|') }}
        -> 1 |2|3

    {{ [1, 2, 3]|join }}
        -> 123
    ```

It is also possible to join certain attributes of an object:

```
{{ users|join(', ', attribute='username') }}
```

- `last`(*seq*): Returns the last item of a sequence.
- `length`(*object*): Returns the number of items of a sequence or mapping.
  Aliases: `count`
- `list`(*value*): Converts the value into a list. If it were a string, the returned list would be a list of characters.
- `lower`(*s*): Converts a value to lowercase.
- `map`(): Applies a filter on a sequence of objects or looks up an attribute. This is useful when dealing with lists of objects, but you are only interested in a certain value of it.
  The basic usage is mapping on an attribute. Imagine you have a list of users, but you are only interested in a list of usernames:

    ```
    Users on this page: {{ users |map(attribute='username')|join(', ') }}
    ```

    Alternatively, you can let it invoke a filter by passing the name of the filter and the arguments afterward. A good example would be applying a text conversion filter on a sequence:

    ```
    Users on this page: {{ titles |map('lower')|join(', ') }}
    ```

- `pprint`(*value, verbose=False*): Pretty print a variable. Useful for debugging. With Jinja 1.2 onwards you can pass it a parameter. If this parameter is truthy, the output will be more verbose (this requires pretty).
- `random`(*seq*): Returns a random item from the sequence.
- `reject`(): Filters a sequence of objects by applying a test to each object, and rejecting the objects whose tests succeed.
  If no test is specified, each object will be evaluated as a boolean. For example:

    ```
    {{ numbers|reject("odd") }}
    ```

- `rejectattr()`: Filters a sequence of objects by applying a test to the specified attribute of each object, and rejecting the objects whose tests succeed.
  If no test is specified, the attribute's value will be evaluated as a boolean.

```
{{ users|rejectattr("is_active") }}
{{ users|rejectattr("email", "none") }}
```

- `replace(`*s, old, new, count=None*`)`: Returns a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced; the second is the replacement string. If the optional third argument count is given, only the firstcount occurrences are replaced:

```
{{ "Hello World"|replace("Hello", "Goodbye") }}
    -> Goodbye World

{{ "aaaaargh"|replace("a", "d'oh, ", 2) }}
    -> d'oh, d'oh, aaargh
```

- `reverse(`*value*`)`: Reverses the object or returns an iterator that iterates over it the other way round.
- `round(`*value, precision=0, method='common'*`)`: Round the number to a given precision. The first parameter specifies the precision (default is `0`), the second the rounding method:
  - `'common'` rounds either up or down: Default method.
  - `'ceil'` always rounds up
  - `'floor'` always rounds down

    ```
    {{ 42.55|round }}
        -> 43.0
    {{ 42.55|round(1, 'floor') }}
        -> 42.5
    ```

  Note that even if rounded to `0` precision, a float is returned. If you need a real integer, pipe it through int:

```
{{ 42.55 |round|int }}
    -> 43
```

- `safe(`*value*`)`: Marks the value as safe which means that in an environment with automatic escaping enabled this variable will not be escaped.
- `select()`: Filters a sequence of objects by applying a test to each object, and only selecting the objects whose tests succeed.
  If no test is specified, each object will be evaluated as a boolean. For example,

```
{{ numbers|select("odd") }}
{{ numbers|select("odd") }}
```

- `selectattr()`: Filters a sequence of objects by applying a test to the specified attribute of each object, and only selecting the objects whose tests succeed.
  If no test is specified, the attribute's value will be evaluated as a boolean. For example,

```
{{ users|selectattr("is_active") }}
{{ users|selectattr("email", "none") }}
```

- `slice(`*value, slices, fill_with=None*`)`: Slices an iterator and returns a list of lists containing those items. Useful if you want to create a div containing three ul tags that represent columns:

```
<div class="columwrapper">
  {% for column in items|slice(3) %}
    <ul class="column-{{ loop.index }}">
    {% for item in column %}
      <li>{{ item }}</li>
    {% endfor %}
```

```
      </ul>
   {% endfor %}
</div>
```

If you pass it a second argument, it is used to fill missing values on the last iteration.

- `sort`(*value, reverse=False, case_sensitive=False, attribute=None*): Sorts an iterable. Per default it sorts ascending, if you pass it true as the first argument, it will reverse the sorting.
  If the iterable is made of strings, the third parameter can be used to control the case sensitiveness of the comparison which is disabled by default.

```
{% for item in iterable|sort %}
    ...
{% endfor %}
```

It is also possible to sort by an attribute (for example to sort by the date of an object) by specifying the attribute parameter:

```
{% for item in iterable|sort(attribute='date') %}
    ...
{% endfor %}
```

- `string`(*object*): Makes a string unicode if it isn't already. That way a markup string is not converted back to unicode.
- `striptags`(*value*): Strips SGML/XML tags and replace adjacent whitespace by one space.
- `sum`(*iterable, attribute=None, start=0*): Returns the sum of a sequence of numbers plus the value of parameter 'start' (which defaults to 0). When the sequence is empty, it returns to start.
  It is also possible, to sum up only certain attributes:

```
Total: {{ items|sum(attribute='price') }}
```

The *attribute* parameter was added to allow summing up over attributes. Also, the *start* parameter was moved on to the right.

- `title`(*s*): Returns a titlecased version of the value, i.e., words will start with uppercase letters, all remaining characters are lowercase.
- `tojson` or `toJSON` (*value, indent=None*): Dumps a structure to JSON so that it's safe to use in `<script>` tags. It accepts the same arguments and returns a JSON string. Note that this is available in templates through the `|tojson` filter which will also mark the result as safe. Due to how this function escapes certain characters this is safe even if used outside of `<script>` tags.
  The following characters are escaped in strings: `<>&'`
  This makes it safe to embed such strings in any place in HTML with the notable exception of double quoted attributes. In that case single quote your attributes or HTML escape also.
  The indent parameter can be used to enable pretty printing. Set it to the number of spaces that the structures should be indented with.
  Note that this filter is for use in HTML contexts only.
- `trim`(*value*): Strips the leading and trailing whitespace.
- `truncate`(*s, length=255, killwords=False, end='...', leeway=None*): Returns a truncated copy of the string. The length is specified with the first parameter which defaults to `255`. If the second parameter is `true`, the filter will cut the text at length. Otherwise, it will discard the last word. If the text was in fact truncated, it would append an ellipsis sign (`"..."`). If you want a different ellipsis sign than `"..."` you can specify it using the third parameter. Strings that only exceed the length by the tolerance margin given in the fourth parameter will not be truncated.

```
{{ "foo bar baz qux"|truncate(9) }}
    -> "foo..."
{{ "foo bar baz qux"|truncate(9, True) }}
    -> "foo ba..."
```

```
{{ "foo bar baz qux"|truncate(11) }}
    -> "foo bar baz qux"
{{ "foo bar baz qux"|truncate(11, False, '...', 0) }}
    -> "foo bar...
```

The default leeway on newer Jinja2 versions is 5 and was 0 before but can be reconfigured globally.

- `upper`(*s*): Converts a value to uppercase.
- `urlencode`(*value*): Escape strings for use in URLs (uses UTF-8 encoding). It accepts both dictionaries and regular strings as well as pairwise iterables.
- `urlize`(*value, trim_url_limit=None, nofollow=False, target=None, rel=None*): Converts URLs in plain text into clickable links.
  If you pass the filter an additional integer, it will shorten the URLs to that number. Also, a third argument exists that makes the URLs "nofollow":

```
{{ mytext|urlize(40, true) }}
    links are shortened to 40 chars and defined with rel="nofollow"
```

  If the *target* is specified, the `target` attribute will be added to the `<a>` tag:

```
{{ mytext|urlize(40, target='_blank') }}
```

- `wordcount`(*s*): Counts the words in that string.
- `wordwrap`(*s, width=79, break_long_words=True, wrapstring=None*): Returns a copy of the string passed to the filter wrapped after `79` characters. You can override this default using the first parameter. If you set the second parameter to *false* Jinja will not split words apart if they are longer than the *width*. By default, the newlines will be the default newlines for the environment, but this can be changed using the `wrapstring` keyword argument.
- `xmlattr`(*d, autospace=True*): Creates an SGML/XML attribute string based on the items in a dict. All values that are neither none nor undefined are automatically escaped:

```
<ul{{ {'class': 'my_list', 'missing': none,
        'id': 'list-%d' |format(variable)}|xmlattr }}>
...
</ul>
```

Results in something like this:

```
<ul class="my_list" id="list-42">
...
</ul>
```

As you can see it automatically prepends a space in front of the item if the filter returned something unless the second parameter is false.

json_query filter

Use the json_query filter when you have a complex data structure in the JSON format from which you require to extract only a small set of data. The `json_query` filter enables you to query and iterate a complex JSON structure. The filter is built using jmespath, and you can use the same syntax in the `json_query` filter. For details on jmespath, see JMESPath Examples.

Example

The result of your playbook step is as follows:

```
[
 {"name": "a", "state": "running"},
```

```
{"name": "b", "state": "stopped"},
{"name": "b", "state": "running"}
]
```

From this result, you want to query only the names of those objects who are in the `running` state. For this query the valid jinja expression would be: `{{ vars.result | json_query(" [?state=='running'].name") }}`, which would have the following result:

```
[
  "a",
  "b"
]
```

To create a valid JSON query, you can refer to JMESPath Tutorial.

Jinja Expressions in FortiSOAR

Following are some examples of jinja expressions used in FortiSOAR:

For Loop

At times, it is useful to use a combination of conditional logic and looping to check particular conditions across a list of values. In this case, the `for` and the `if` loop is useful in the Dynamic Variable language.

In the following example, an evaluation of assignment is run across a specific dictionary representing all associated teams within a particular record. These teams have already been assigned to a particular variable from the parent entity:

```
{% for item in vars.teamName %}
      {% if item == '/api/3/teams/97fd5a3f-4eaf-4cc1-a132-1c9274bd8428' %}
      Yes  {% endif %}
{% endfor %}
```

If Condition

```
{% if 1485561600000 > 1484092800000 %}
      {{vars.input.records[0]}}
          {% elif 5==6 %}
      {{vars.input.records[0]}}
{% endif %}
```

An `if` condition can cause a playbook to fail if the jinja that you have added returns an empty string, which is not compatible with the field datatype defined in the database. For example, if you have added the following jinja to a `{% if vars.currentValue == "Aftermath" %}{{@Current_Date}}{% endif %}` field, then the playbook will fail if the jinja returns an empty string.

To ensure that your playbook does not fail due to the issue of jinja returning an empty string, add the following jinja in the field: `{% if vars.currentValue == "Aftermath" %}{{Current_Date}}{% else %} None {% endif %}`.

For Loop along with the If condition

At times, it can be useful to use a combination of conditional logic and looping during a check of particular conditions across a list of values. In this case, the `for` and the `if` loop is useful in the Dynamic Variable language.

```
{% for i in vars.var_list  %}
        { %  if i not in vars.var_response % }
            {{vars.var_response.append(i)}}
        {% endif %}
{% endfor %}
```

The following example uses the `for` and the `if` loop to check if a particular team is tagged to a record. In the following example, an evaluation of assignment is run across a specific dictionary representing all of the associated teams within a particular record. These teams have already been assigned to a particular variable from the parent entity.

```
{% for item in vars.teamName %}
        {% if item == '/api/3/teams/97fd5a3f-4eaf-4cc1-a132-1c9274bd8428' %}
        Yes  {% endif %}
{% endfor %}
```

If Else condition

`If` conditions within the Dynamic Variable templating engine can be very useful to avoid unnecessary decision steps. These conditions allow you to specify values defined within specific conditions such that copying, or value branches are not needed.

The following example uses the if else condition to create mapping between Alert Severity and Incident Category.

Here is an example of a particular case in which the value of an IRI is defined based upon the specific conditions evaluated during the execution. Bear in mind these IRI values must be known in this case.

```
{% if vars.alertSeverity == "/api/3/picklists/ad5eacc1-7c05-3c4e-bba7-eb356c8547c9" %}

        /api/3/picklists/58d0753f-f7e4-403b-953c-b0f521eab759

{% elif vars.alertSeverity == "/api/3/picklists/ddbc7842-dde0-392a-ae94-a1e7a3c7c2f7" %}

        /api/3/picklists/40187287-89fc-4e9c-b717-e9443d57eedb

{% elif vars.alertSeverity == "/api/3/picklists/6c7a8653-a2d5-3611-bc86-8ca307a02e88" %}

        /api/3/picklists/7efa2220-39bb-44e4-961f-ac368776e3b0

{% endif %}
```

Time Operations

To get timestamp:

```
{{ arrow.get('2013-05-30 12:30:45', 'YYYY-MM-DD HH:mm:ss') }}
```

To convert current time into epoch and multiply by 10000:

```
{{arrow.utcnow().timestamp*1000 |int| abs}}
```

To convert date to epoch time:

```
{{ arrow.Arrow(2017, 3, 30).timestamp}}
```

Convert timezone from UTC to any with formatting

Use the Arrow library to convert dates and times from one-time zone/format to another.

The general format is as follows.

```
{{ arrow.get( % VARIABLE % ).to('% TIME ZONE ACRONYM %').format('% FORMAT STRING%') }}
```

The following is an example that is converting the Date of Compromise field into a readable Eastern Standard Time format.

```
{{arrow.get(vars.input.records[0].dateOfCompromise).to('EST').format('YYYY-MM-DD HH:mm:ss
ZZ')}}
```

Convert timezone from any to UTC and move it up

Using the Arrow library, the general format is as follows.

```
{{ arrow.get( % VARIABLE %).replace( % TIME VALUE % = % OPERATOR + VALUE %) }}
```

An example where the Alert Time value is replaced with a four-hour increase.

```
{{ arrow.get(vars.alertTime).shift(hours=+4) }}
```

Convert to epoch time to insert into a Record

In this example, a particular UTC time is converted into epoch time in order to format it for API insertion. The API will accept times in epoch as the default.

```
{{ arrow.get(vars.utcTime).timestamp }}
```

String Operations

To find the length of list or string:

```
{{vars.emails | length }}
```

To replace a string:

```
{{ vars.var_keys.replace("dict_keys(","" ) | replace( ")", "" )}}
```

Strip the first X characters of a string

Starting with a string variable, you can pull a portion of the string based on counting the characters.

The general format for this Jinja expressions is as follows where # is the number of characters.

```
{{ % VARIABLE %[:#] }}
```

An example here pulls the first 17 characters of the string `timeRange`.

```
timeRange = 2016-09-02 13:45:00 EDT - 2016-09-02 14:00:00 ET
alertTime = {{vars.timeRange[:17]}}

print(alertTime)

2016-09-02 13:45:00
```

Remove Strip tags

A particularly useful filter within Dynamic Variables is the `striptags()` function. This allows you to remove the HTML tags on a particular value, which may be present in rich text or other HTML formats.

The function preserves the data contained within the tags and removes any tagged information contained around the actual values.

```
{{ vars.input.records[0].name.striptags() }}
```

Code in block

```
{% block body %}
        {% for key, value in vars.loop_resource.items() %}
            {{ key }}: {{ value }}
        {% endfor %}
{% endblock %}
```

Set variable based on condition

```
{% for i in vars.result['hydra:member'] %}
        {% set id = i['@id'] %}
        {{ vars.inc_fdata.append(id) }}
{%endfor%}
```

Second example:

```
{% for i in vars.result['hydra:member'] %}
        {% set id = i['@id'] %}
        {% set createDate = i.createDate | string %}
        {% set list_item = [id,createDate] %}
            {{ vars.inc_fdata.append(list_item) }}
{%endfor%}
```

Custom Functions and Filters

FortiSOAR supports following custom functions/filters:

- `Get_current_date`: Returns the current date for the file.
- `Get_current_datetime`: Returns the current date and time for the file.
- `currentdateminus`: Returns a timestamp value of the current date minus the specified days from the current date.

For example, `{{ currentDateMinus(10) }}` returns a timestamp after deducting 10 days from the current date.

- `uuid`: Returns the UUID of the file `{{ uuid() }}`.
- `arrow`: Returns a python arrow library.
- `toJSON`: Converts a JSON to a string. Useful for storing a JSON in a FortiSOAR textarea field, for example, `Source Data`, so that JSON renders correctly and the content can be presented nicely in the UI.
- `html2text`: Converts an HTML string to a text string.
  `{{ html_string | html2text}}`
  For example, `{{'<br>this is html text </br>' | html2text}}`.
  Output will be - this is html text.
- `json2html`: Converts JSON data into HTML. The FortiSOAR template is used for HTML and styling of the output.
  `{{ jsondata | json2html(row_fields)}}`
  `row_fields`= `['pid', 'sid']`. You can optionally specify the row_fields attribute. If you do not specify the row_fields, by default, this filter takes all keys as row fields.
  An example without row fields specified: `{{ [{"pid": 123, "sid": "123", "path": "abc.txt"}] | json2html}}`.
  An example with row fields specified: `{{ [{"pid": 123, "sid": "123", "path": "abc.txt"}] | json2htmll(['pid', 'sid'])}}`.
  The HTML output of the above example will be:
  ```
  <table class="cs-data-table"> <tr> <th>pid</th> <th>sid</th> </tr> <tr>
  <td>123</td> <td>123</td> </tr> </table><button style="display:none" class="cs-
  datatable-btn btn-link cs-datatable-showmore-btn" type="button"
  onClick="event.target.previousElementSibling.className += ' cs-data-table-show-
  more'; event.target.nextElementSibling.style.display = 'block';
  event.target.style.display = 'none';">Show more</button><button class="cs-
  datatable-btn btn-link cs-datatable-showless-btn" type="button"
  onClick="event.target.previousElementSibling.previousElementSibling.className =
  'cs-data-table'; event.target.previousElementSibling.style.display = 'block';
  event.target.style.display = 'none';">Show less</button>
  ```
- `resolveIRI` : Resolves the given IRI.
  For example, `{{ "/api/3/alerts/<alert-id>" | resolveIRI}}`
- `count_occurrence`: Retrieves the number of times each element appears in the list.
  For example, `{{ ['apple','red','apple','red','red','pear'] | count_occurrence }}`
  The output of this example is: `{'red': 3, 'apple': 2, 'pear': 1}`
- `urlencode`: Encodes the given URL.
  For example, `{{"/api/3/alerts/?name=test" | urlencode}}`
  The output of this example is: `%2Fapi%2F3%2Falerts%2F%3Fname%3Dtest`
- `urldecode`: Decodes the given (encoded) URL.
  For example, `{{"%2Fapi%2F3%2Falerts%2F%3Fname%3Dtest" | urldecode}}`
  The output of this example is: `/api/3/alerts/?name=test`
- `loadRelationships(moduleName, selectFields = [])`: Used to fetch details of a related (correlation) record. For example, `{{ vars.incidentIRI | loadRelationships('indicators') }}`
  To fetch complete details of the correlation record, use `{{ #recordIRI# | loadRelationships ('#CorrelationModuleFieldName#') }}`
  To fetch specific fields of the correlation record, use `{{ #recordIRI# | loadRelationships ('#CorrelationModuleFieldName#',['#field1#','#field2#']) }}`
- `picklist`: Loads the specified picklist item object. For example, `{{"PicklistName" | picklist ("ItemValue") }}`
  The output of this example is an object including the `@id`, `color`, `itemValue`, `listName`, and `orderIndex` of the picklist item. You can extract just a particular key from the object by specifying a second argument to the filter:

```
{{"PicklistName" | picklist("ItemValue", "@id") }}.
```
This will generate
```
/api/3/picklists/<uuid>
```

Debugging and Optimizing Playbooks

This chapter explains how you can easily debug playbooks in FortiSOAR using execution history and executed playbooks logs. It also provides you information on how to tune various keys and troubleshoot playbook errors.

> The Integrations API call has been changed in version 7.0.0 to support only `POST` calls; earlier `GET` calls were also supported. Therefore, if you have any existing playbooks that uses the `GET` calls, then that playbook will fail. To resolve this issue, you have to manually change the method from `GET` to `POST` in your playbooks.

From version 7.0.2 onwards, you can define the logging levels (`INFO` or `DEBUG`) for your playbook execution logs, both globally as well as at the individual playbook level. For more information on playbook logging levels, and how to set those levels on individual playbooks, see the Introduction to Playbooks chapter. Note that the 'Debugging Playbooks' content assumes that you are running the playbooks in the `DEBUG` mode.

Debugging Playbooks

As you develop more sophisticated Playbooks, the ability to easily debug playbooks becomes exceeding important. FortiSOAR has designed the Execution History to make it easier for you to see the results of your executed playbooks and for you to debug playbooks.

Use the **Executed Playbook Logs** icon (   ) that appears on the top-right corner of the FortiSOAR screen to view the logs and results of your executed playbooks as soon as you log on to FortiSOAR. You can also use the executed playbook logs to debug your playbooks.

> FortiSOAR implements Playbook RBAC, which means that you can view logs of only those playbooks of which you (your team) are the owner. For more information, see the Introduction to Playbooks chapter.

The Execution History provides the following details:

- Playbooks have been organized by the parent-child relationship.
- Playbooks have a console using which you can see debug messages with more significant details.
- Playbook designer includes the playbook execution history option.
- Playbooks can be filtered by Playbook Name or Record IRI, user, date range, or status.
- Playbook Execution History contains details of the playbook result, including information about the environment and the playbook steps, including which steps were completed, which steps are awaiting some action, which steps were failed, and which steps were skipped.

> From version 7.0.2 onwards, users will not be able to view the execution history of 'legacy' playbooks, i.e. the execution history of playbooks that were run before release 6.0 will not be visible.

The Executed Playbook Logs do not display the Trace information from the error message so that the readability of the Executed Playbook Logs is enhanced since the clutter in the error details screen is reduced and you can directly view the exact error. The Trace information is yet present in the playbook logs.

FortiSOAR also contains enhanced the error messages that are precise and detailed making it easier for you to debug playbook issues. For information about the common playbook error messages and how to debug them, see the Debugging common playbook and connector errors article present in the Fortinet Knowledge Base.

You can access the playbook execution history as follows:

- Clicking the **Executed Playbook Logs** icon ( ) in the upper right corner of the FortiSOAR screen. You have an option of purging executed playbooks logs from the `Executed Playbooks Log` dialog. For more information see Purging Executed Playbook Logs.

- Clicking **Tools** > **Execution History** in the playbook designer to view the execution history associated with that particular playbook.
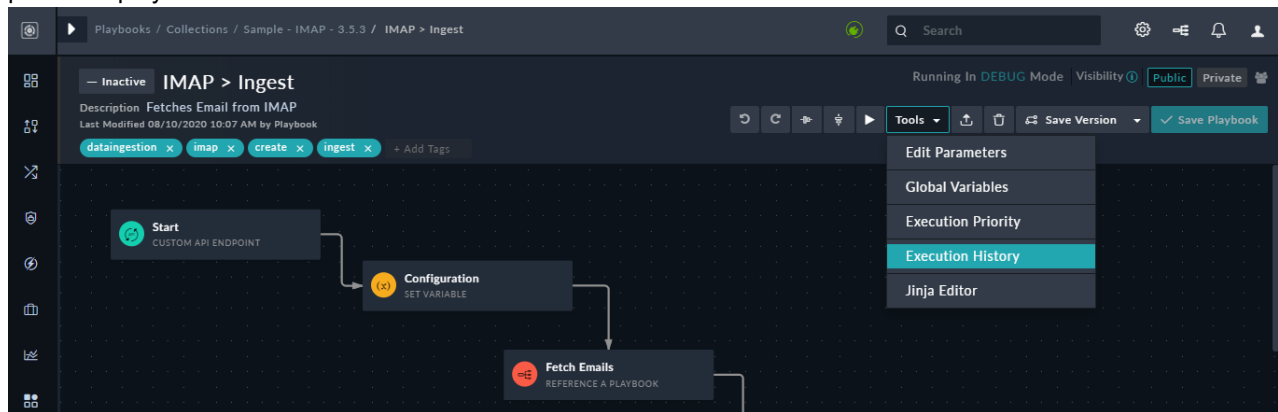
- Clicking the **Executed Playbook Logs** icon in the detail view of a record such as an alert record to view the playbooks that have been executed on that particular record in a flowchart format. This makes it easier for users to view the flow of playbooks, especially useful for viewing the parallel execution paths in playbooks.
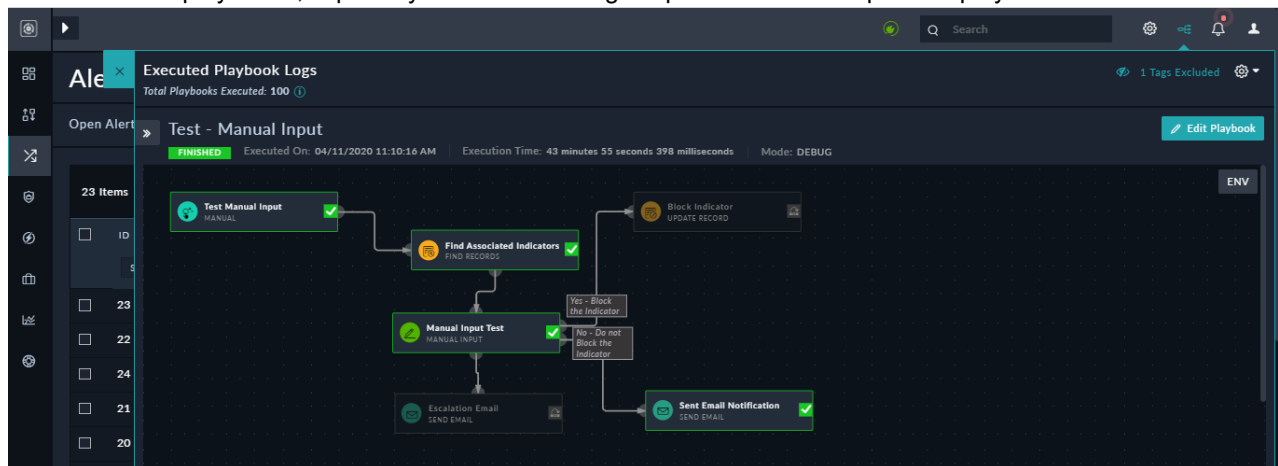
Playbook Execution History

Click the **Executed Playbook Logs** icon in the upper-right corner of FortiSOAR to view the logs and results of your executed playbook. Clicking the **Executed Playbook Logs** icon displays the Executed Playbook Logs dialog as shown in the following image:
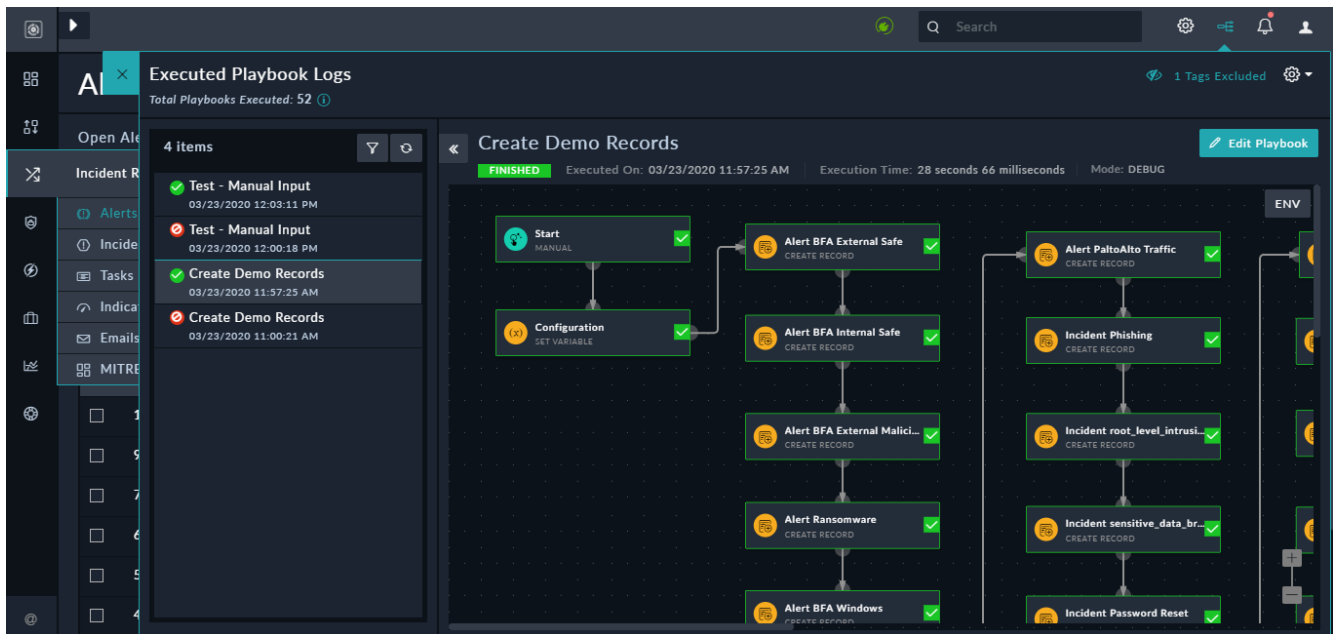
The **Executed Playbook Logs** displays the executed playbooks in the flowchart format, as is displayed in the playbook designer. This makes it easier for users to view the flow of playbooks, especially useful for viewing the parallel execution paths in playbooks.

Playbooks are sorted by chronological datetime, with the playbook that was executed last being displayed first. All playbooks are displayed with **10** playbooks being displayed per page. Click a playbook in the list to display it in the flowchart format and also see the details of the playbook result, the environment and the playbook steps, including which steps are completed, failed, awaiting or skipped.

The Execution Playbook Log dialog also displays a count of the total playbooks executed, the date time of when the playbook was executed, and the time taken for executing the playbook.

When you click on playbook steps, you can toggle the **ENV** button to toggle between the environment of the step and the output of the step. You can also copy the environment, error, and step details to the clipboard by clicking the **Copy 'Env' to Clipboard** or **Copy 'OUTPUT' to Clipboard** button.

You can also open the playbook directly in the playbook designer from the **Executed Playbook Logs** dialog by clicking the **Edit Playbook** button that appears in the right section of the dialog.

You can collapse and expand the **Executed Playbook Logs** dialog by clicking the **<<** or **>>** arrows as shown in the following image:
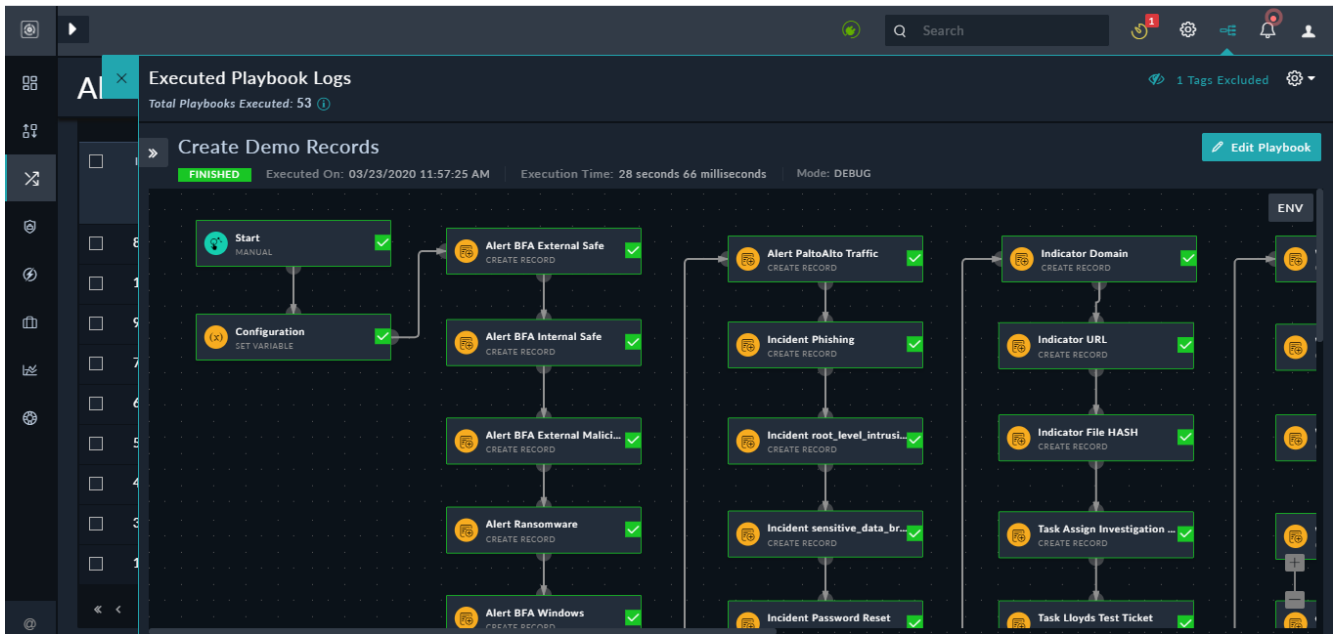
You can refresh the playbook logs and filter logs associated with playbooks using the **Filter** icon:
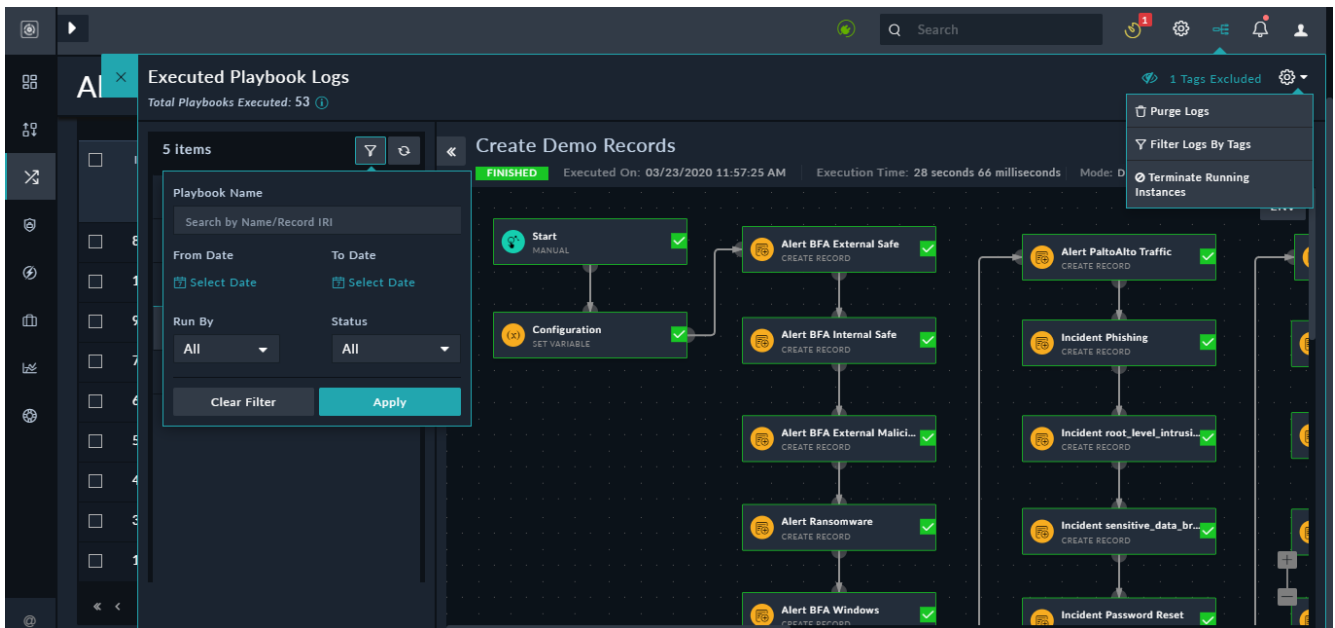
Clicking the **Filter** icon allows you to filter playbook logs using the following options:

- **Playbook Name**: In the **Search by Playbook Name or Record IRI** field, filter the log associated with a particular playbook, based on the playbook name or the record IRI associated with the playbook.
  Example of filtering logs using the Record IRI: `/alerts/bd4bf0a6-b023-4bd7-a182-f6938fa37ada`.

- **From Date**: You can filter the log based on the date from which the playbooks were executed.

- **To Date**: You can filter the log based on the date till when the playbooks were executed. Using the From Date and To Date fields, you can create a data range for retrieving the logs of playbook executed during that time period.

- **Run By**: From the **Run By** drop-down list, filter the log associated with a particular playbook, based on the user who has run the playbook.

- **Status**: From the **Status** drop-down list, filter the log associated with a particular playbook, based on the status of the playbook execution. You can choose from the following options: Incipient, Active, Awaiting, Paused, Failed, Finished, or Finished with error.

You will also see the timestamp when the playbook was executed and the time it took for the playbook to complete its execution.

To purge Executed Playbook Logs, click the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog and select the **Purge Logs** option. For more information, see Purging Executed Playbook Logs.

To terminate a playbook that are in the **Active**, **Incipient**, or **Awaiting** state, click the Terminate button. To terminate all running instances of a particular type, click the **Settings** icon and select the **Terminate Running Instances** option. For more information, see Terminating playbooks.

Environment

Click **Env** to view the complete environmental context in which the playbook was executed, including the input-output and computed variables across all steps in the playbook.
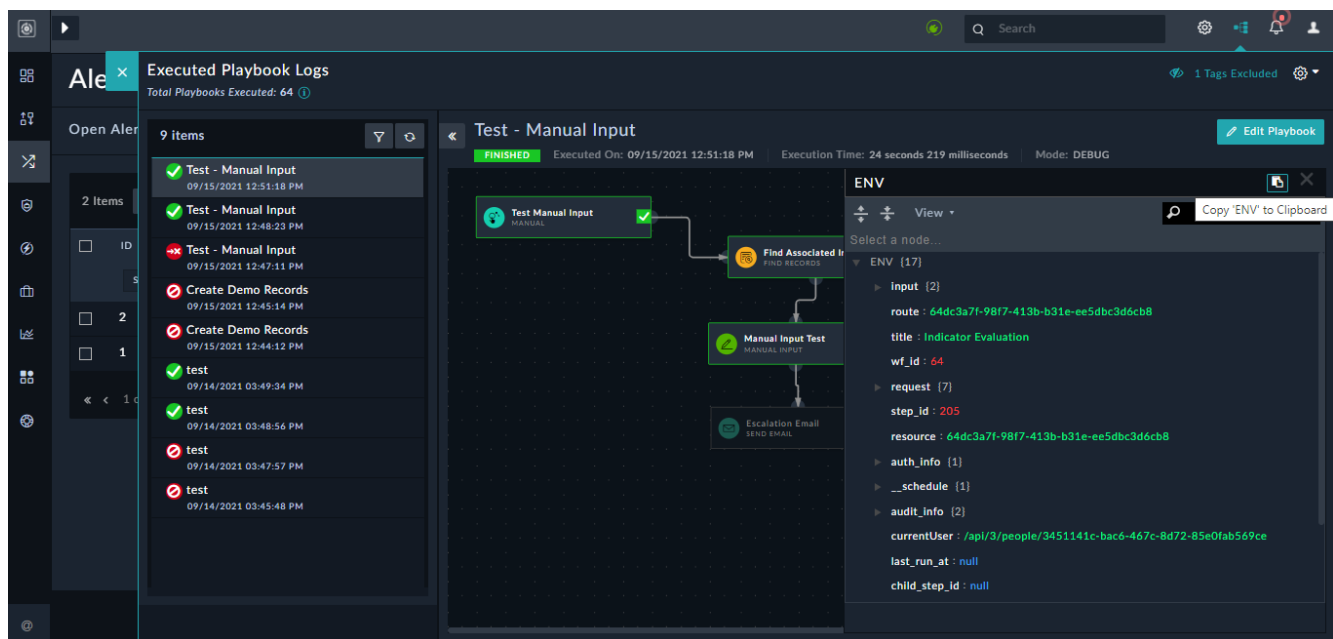
Running huge ingestions and other workflows that load several records into memory can cause the memory usage to be high. In case of connector actions, the 'env' was passed on to each action and it also used to get the 'env' triggered from the workflow (worker 'env'). This unnecessarily increases the memory requirement and limits workflow scaling. Also, connectors only need a minimal information such as `requests headers`, `public-private key`, and `auth info` from the 'env' that can be selectively passed. Therefore, to solve the memory consumption issue, connectors are passed only the required 'env' fields and not the complete environment information. However, if you observe any issues in a connector or you specifically require the complete environment to be passed to the connector, then you need to add the `CONNECTOR_KEEP_COMPLETE_ENV` variable at the end of the `/opt/cyops-workflow/sealab/sealab/settings.py` file, and save the file. Then, you must restart the `celeryd` service using the following command:
```
# systemctl restart celeryd
```

Playbook Steps

Clicking playbook steps display the input, output, and configuration for that step. You can toggle the **ENV** button to toggle between the environment in which the playbook was executed and the steps of the playbook. You can also copy the environment, error, and step details to the clipboard by clicking the **Copy 'ENV' to Clipboard** or **Copy 'OUTPUT' to Clipboard** button.

Clicking playbook Steps section lists all the steps that were part of the playbook and displays the status of each step using icons. The icons indicate whether the step was completed (green tick), skipped (grey skipped symbol), awaiting some action (orange hour glass symbol) or failed (red failed symbol).

For example, if a playbook is awaiting some action, such as waiting for approvals from a person or team who are specified as approvers, then the state of such playbooks is displayed as **Awaiting**.

The status of the playbook will display as "Awaiting" till the action for which the playbook execution halted is completed, after which the playbook will move ahead with the workflow as per the specified sequence.

You can click on a playbook step for which you want to view the details, and you will see tabs associated with the playbook step: **Input**, **Pending Inputs** (if the playbook is in the awaiting state), **Output** (if the playbook finishes) or **Error** (if the playbook fails), and **Config**.

Input Tab

The input tab displays data, in the case of the first step of the playbook such as the `Start` step, input arguments and evaluated arguments. The `data` displays the trigger information for the playbook. The `input_args` displays the input in the jinja format that the user has entered for this step. The `evaluated_args` displays what the user input was evaluated by the playbook once the step gets executed.
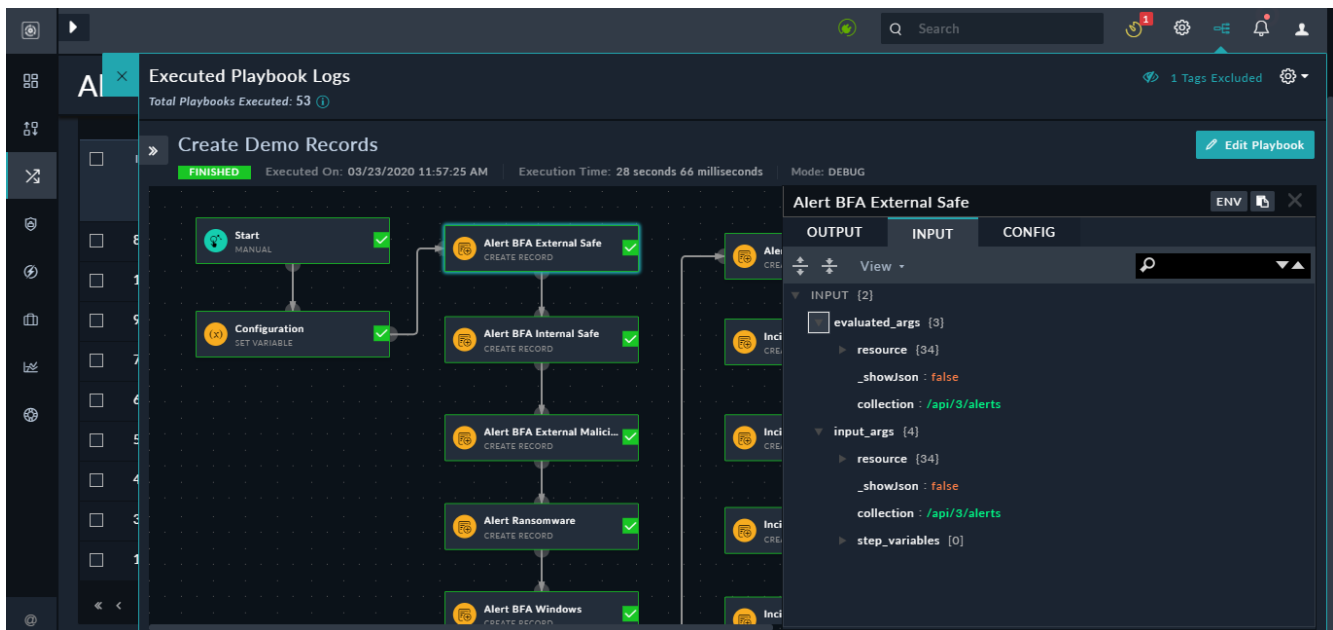
Pending Inputs Tab

If a playbook is in an "Awaiting" state, i.e., it requires some input or decision from users to continue with its workflow, then the Pending Inputs tab is displayed:
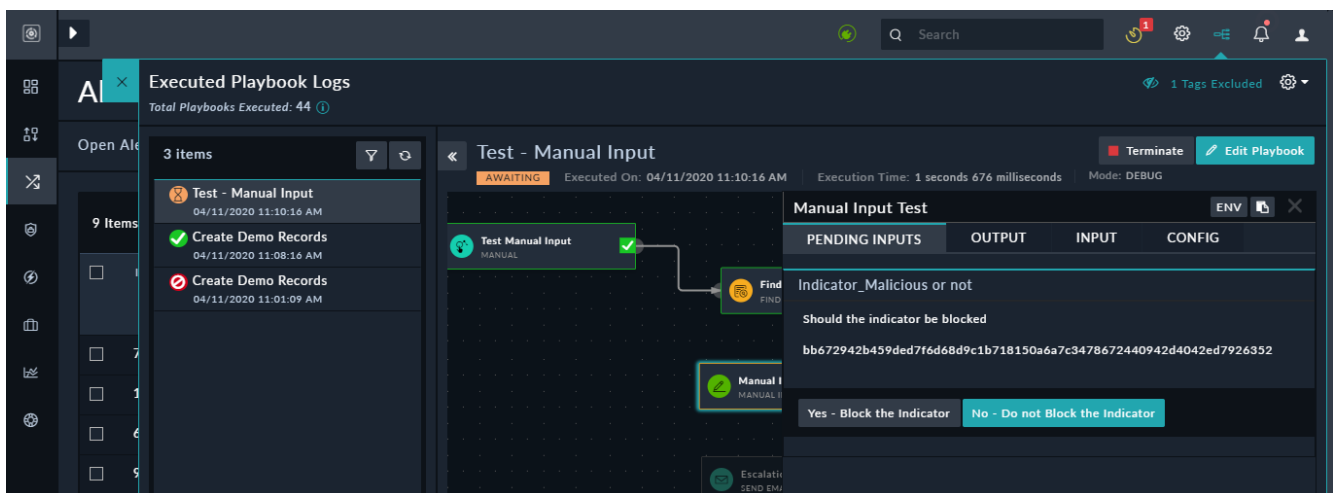
Once the user provides the required inputs and submits their action, the playbook continues its execution as per the defined workflow.

Output or Error Tab

If the playbook step finishes, then the **Output** tab displays the result/output of the playbook step.
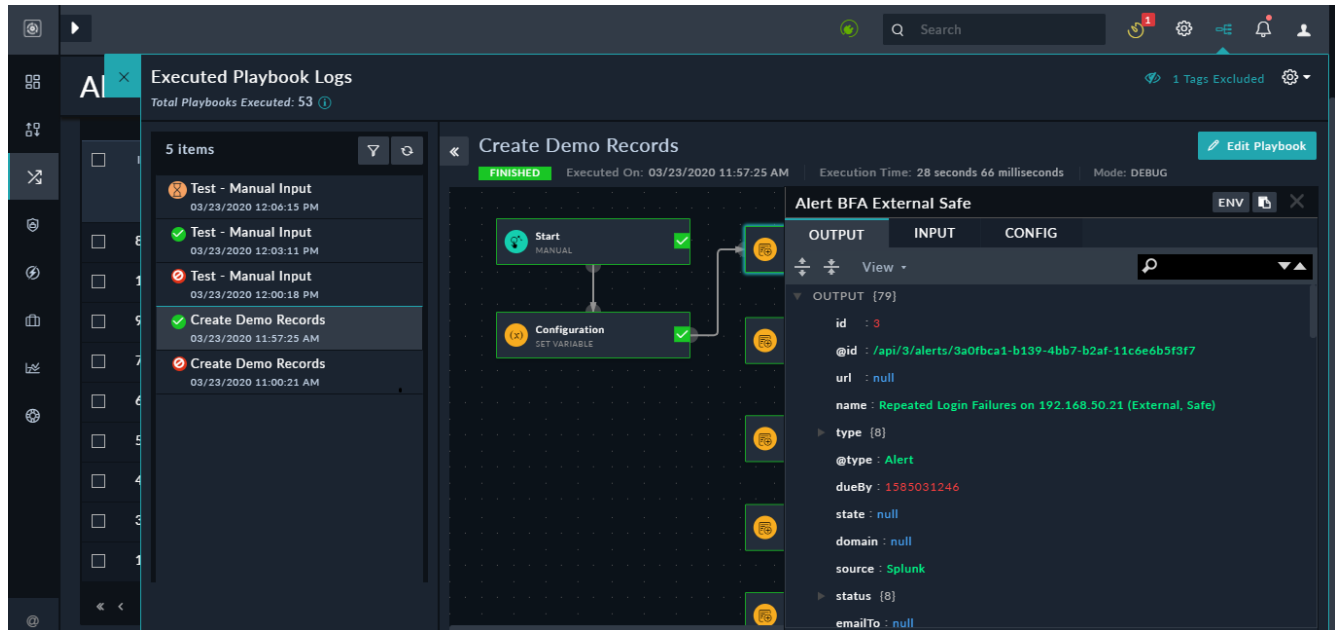
If the playbook step fails, then the **Error** tab displays the **Error message** for that step. Click the step that has the error (step with a red cross icon) to view the error message, so that it becomes easier for you to know the cause of the error and debug the cause of the playbook failure.
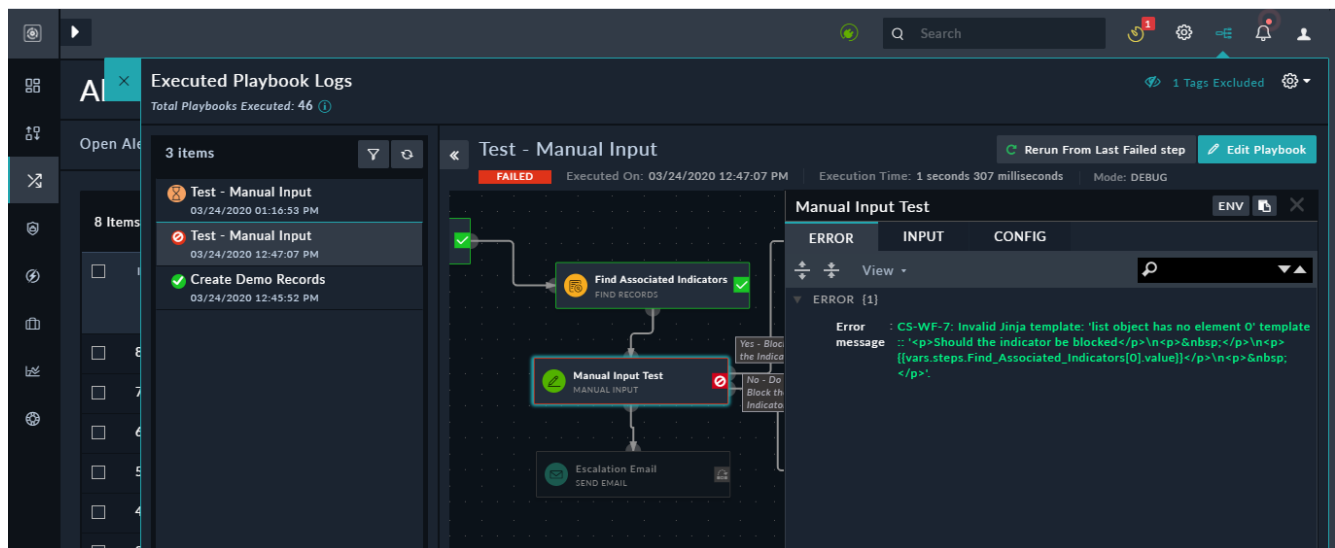
FortiSOAR has enhanced error messages by making them more precise and thereby making it easier for you to debug the issues. Also, the Trace information has been removed from the executed playbook log to reduce the clutter in the error details screen and directly display the exact error. The Trace information will be present in the product logs located at:

- For Playbook runtime issues: `/var/log/cyops/cyops-workflow/celeryd.log`
- For connector issues in cases where playbooks have connectors: `/var/log/cyops/cyops-integrations/connectors.log`

For information about the common playbook error messages and how to debug them, see the Debugging common playbook and connector errors article present in the Fortinet Knowledge Base.

FortiSOAR also provides you with the option to resume the same running instance of a failed playbook from the step at which the playbook step failed, by clicking the **Rerun From Last Failed Step** button. This is useful in cases where the connector is not configured or you have network issues that causes the playbook to fail, since you can resume the same running instance of the playbook once you have configured the connector or resolved the network issues. However, if you change something in the playbook steps, then that would be a rerun of the playbook and not a resume or retry of that playbook.

Users who have `Execute` and `Read` permissions on the `Playbooks` module can rerun playbooks in their own instance. Administrative users who have `Read` permissions on the `Security` module and `Execute` and `Read` permissions on the `Playbooks` module can rerun their own playbooks and also playbooks belonging to users of the same team.

Notes:

- If you have upgraded your FortiSOAR system, then you can resume only those playbooks that were run after the upgrade.
- If you have a playbook that had failed before you upgrade your FortiSOAR system, and post-upgrade you try to resume the execution of that playbook, then that playbook fails to resume its execution.

To resume the running instance of a failed playbook, do the following:

1. Open the `Executed Playbook Logs` dialog.
2. Click the failed playbook that you want to resume, and then click the **Rerun From Last Failed Step** button. FortiSOAR displays the `Playbook retriggered from last failed step` message and the failed playbook resumes from the failed step:
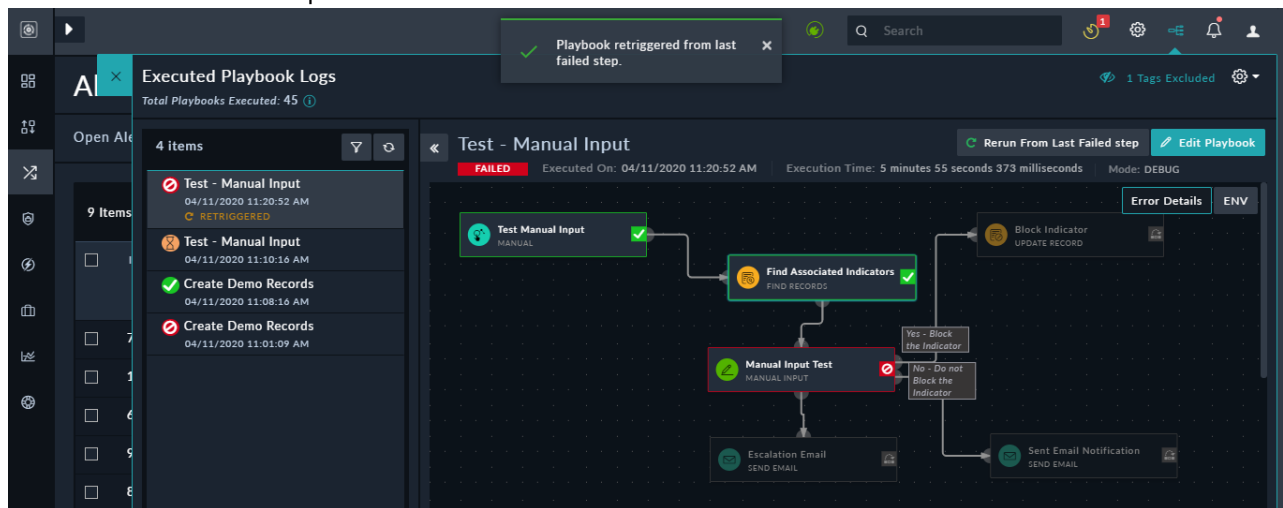
A playbook that has been rerun will display **Retriggered** as shown in the following image:
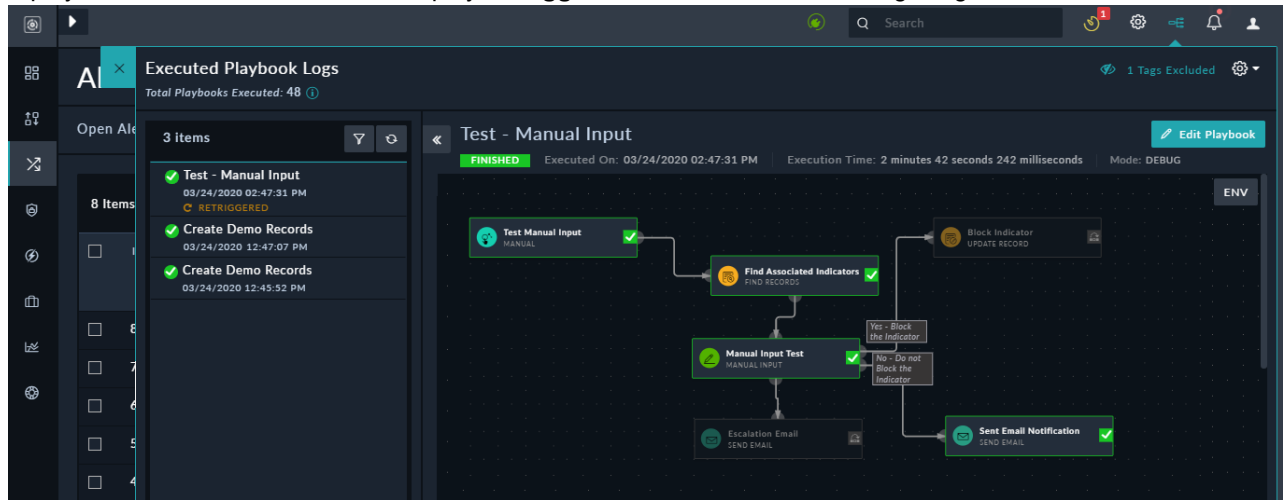
Config Tab

The **Config** tab displays the step variables detailed entered by the user for the particular step and also includes information about whether other variables, such as `ignore_errors`, `MockOutputUsed`, or the `when` condition have been used (`true`/`false`) in the playbook step.

Instances Tab

The **Instances** tab is displayed in case of playbooks containing reference playbook steps. The "Instances" tab allows users to see details such as, name and status of all child instances in a single view.

For example, as displayed in the following image, the "Testing Reference Playbook" references "child 1", which in turn references 10 other child playbooks because of the loop applied on the reference playbook step. When you click the "child 1" step, you can see the "Instances" tab, containing the status of the step "Finished", names of all its child playbooks, and the name of the step that referenced the playbook, which is "Reference to Child 2":
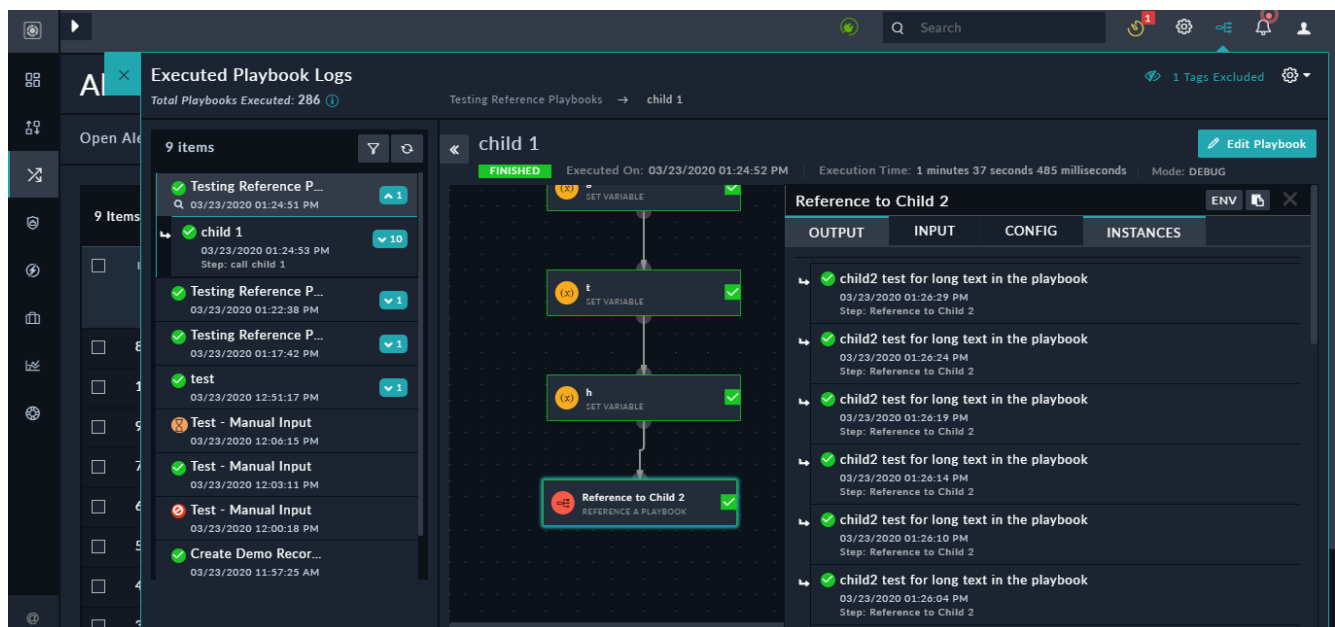
Link to Child Playbooks

Playbooks have been organized by the parent-child relationship. The Parent playbook displays a link that lists the number of child playbook(s) associated with the parent playbook. Clicking the link displays the execution history for the child playbook(s).

The UI of the execution playbook log displays playbooks that contain various levels of child playbooks in the same visual execution log window. You can click the parent playbook and view its child playbooks, and similarly you can view the children of the child playbook by clicking the child playbook all without losing context of the playbook. You can also see the breadcrumb navigation from parent playbook to the child playbook at the top of the playbook log.

The **Executed Playbook Logs** displays the execution history of the child playbooks, i.e., you can search for the child playbook in Executed Playbooks Logs and the search results will display the child playbook and its execution history and you can also use the Load Env JSON feature in the Jinja Editor making debugging of the child playbooks easier.

> Child playbooks inherit the logging level setting from their parent playbook, irrespective of their own setting. For example, if the child playbook's logging level is set as `INFO` and its parent's is set at `DEBUG`; the child playbook's logging level will automatically be set at the `DEBUG` level. For more information on playbook logging levels, and how to set those levels, see the Introduction to Playbooks chapter.

If the parent playbook has a number of child playbooks, you can also search for child playbooks, by clicking the **search** icon that is present beside the child playbook link and then entering the name of the playbook in the **Search by Playbook Name** field. You can also filter the child playbooks on its running status, such as Incipient, Active, Awaiting, etc. by selecting the status from the **All Status** drop-down list.

For example, in the following image, the **Testing Reference Playbook** playbook has 1 child playbook: **child 1**. You can click **child 1** to view its execution history:

As displayed in the above image, you can also see the breadcrumb navigation from parent playbook to the child playbook at the top of the playbook log. You can also view the name of the step at which the child playbook is referenced in the navigator panel. If the playbook contains a reference playbook step then you can click through the child playbooks

within the same visual execution log window, allowing you to navigate through the playbook, without losing the context of the playbook. You can also easily navigate back to the parent playbooks using the breadcrumbs present on top of the log.

For example, as displayed in the following image, the "Testing Reference Playbook" contains a child playbook "child 1" at step "call child 1", which in turn contains 10 other child playbooks because of a loop applied on the reference playbook step, such as "child 2 test for long text in playbook", which in turn calls "child 3":
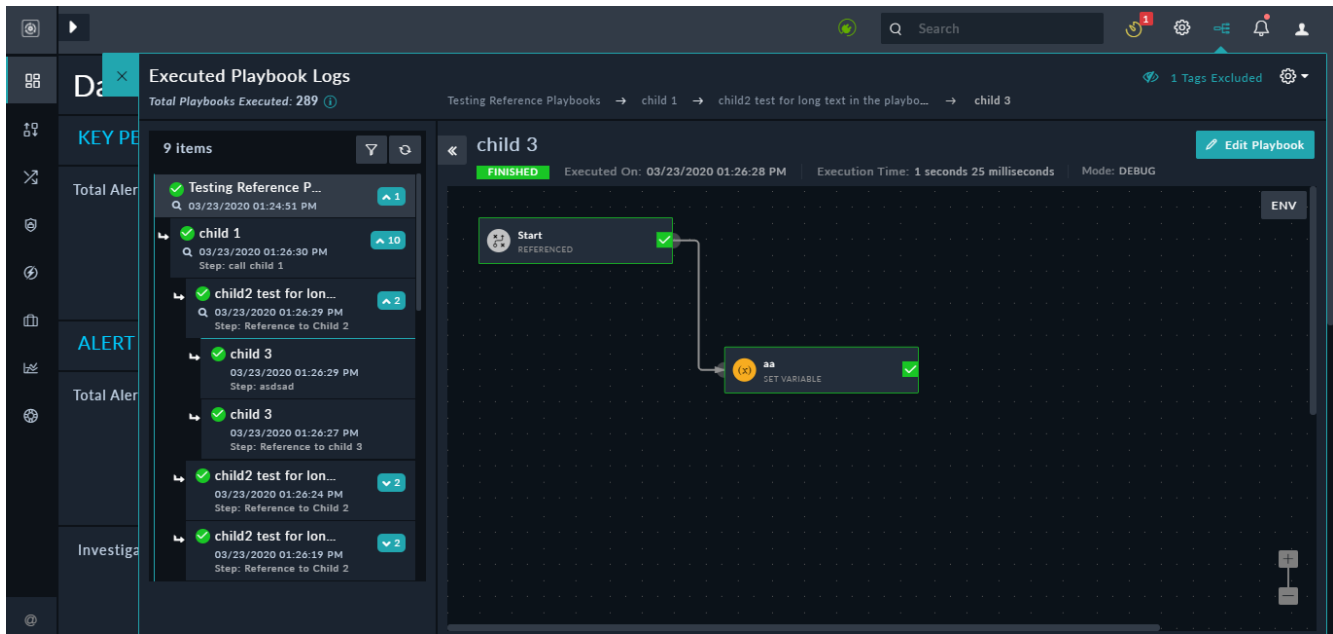
Purging Executed Playbook Logs

You can purge Executed Playbook Logs by clicking the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog, and then selecting the **Purge Logs** option. Purging executed playbook logs allows you to *permanently* delete old playbook history logs that you do not require and frees up space on your FortiSOAR instance. You can also schedule purging, on a global level, for both audit logs and executed playbook logs. For information on scheduling Audit Logs and Executed Playbook Logs, see the `Purging of audit logs and executed playbook logs` topic in the *System Configuration* chapter of the "Administration Guide."

To purge Executed Playbook Logs, you must be assigned a role that has a minimum of `Read` permission on the `Security` module and `Delete` permissions on the `Playbooks` module.

To purge Executed Playbook Logs, click the **Settings** icon and select the **Purge Logs** option, which displays the `Purge Playbook Execution Logs` dialog:
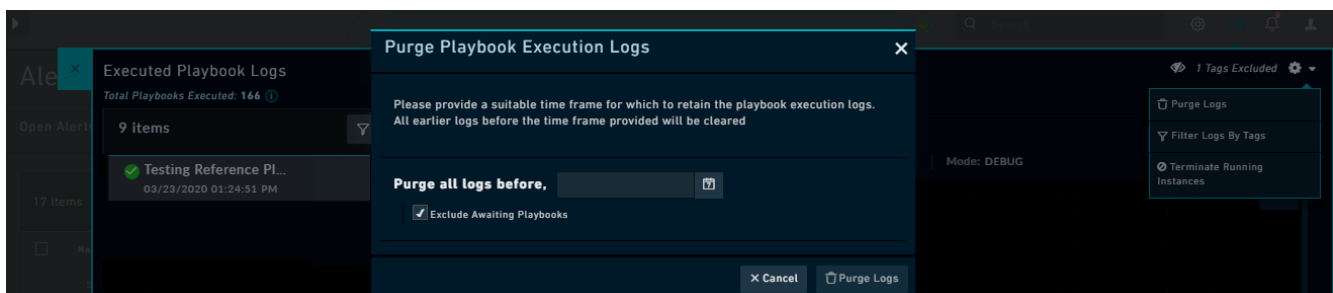
In the **Purge All logs before,** field, select the time frame (using the calendar widget) before which you want to clear all the audit logs. For example, if you want to clear all audit logs before `December 01st, 2019, 9:00 AM`, then select this date and time using the calendar widget.
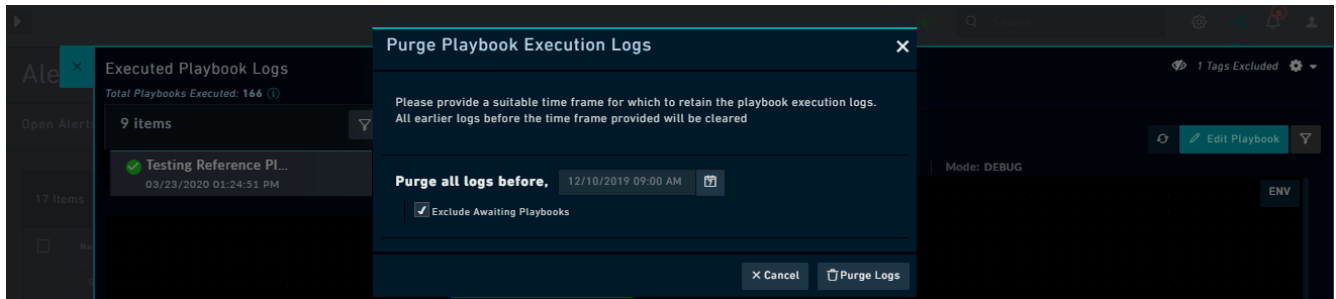
Click the **Exclude Awaiting Playbooks** checkbox (default) to exclude the playbooks that are in the "Awaiting" state from the purging process.

To purge the logs, click the **Purge Logs** button, which displays a warning as shown in the following image:
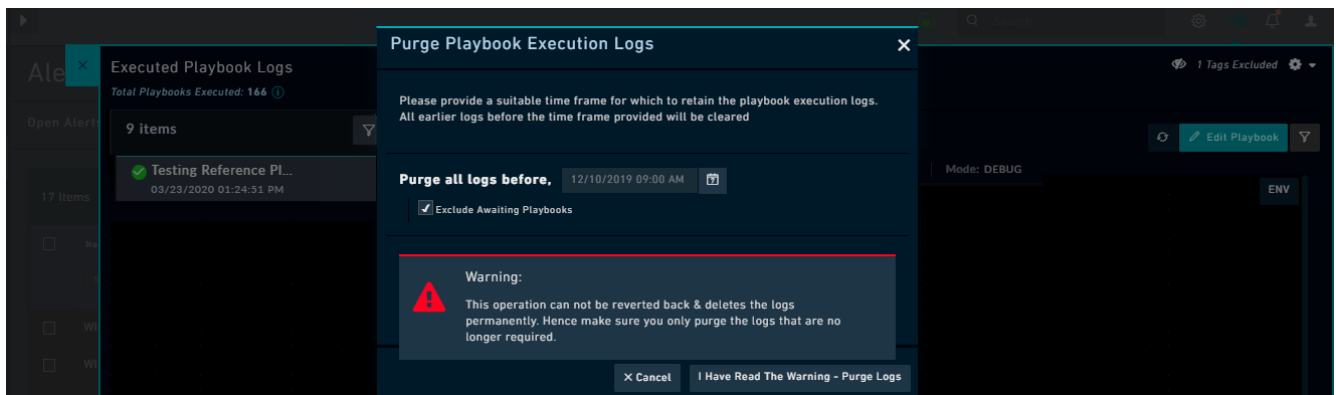
Click the **I Have Read the warning - Purge Logs** to continue the purging process.

Filtering playbook logs by tags

You can filter playbook execution logs by tags or keywords that you have added in your playbooks.

A user who has a role with a minimum of `Update` permission on the `Security` module can save tags, which will be applied as a default filter for playbook execution logs to all other user. A user who does not have such a role can add a tag to filter playbook execution logs and view the filtered playbook execution logs but cannot save that filter.

Click the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog to view tags that have been added by default to filter the playbook execution logs. You can see a message `1 Tags Excluded`, which means that playbook logs with one specific tag is being excluded by default.
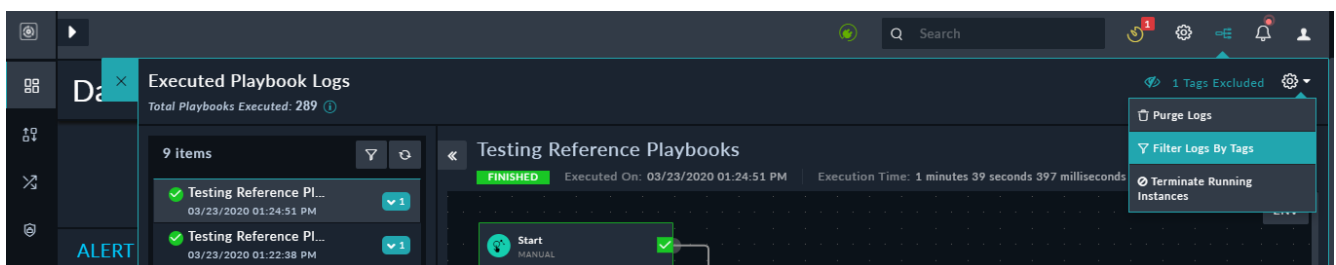
You can either click the **1 Tags Excluded** link or the **Filter Logs By Tags** option to open the `Filter Logs by Tags` popup as shown in the following image:
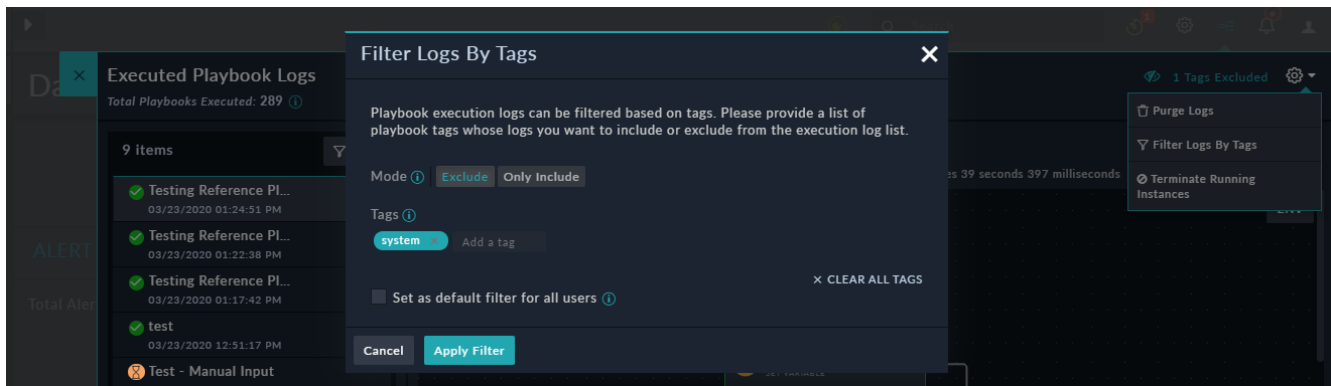
To filter playbook logs based on tags, add a comma-separated list of tags in the **Tags** field.

In the `Mode` section, choose **Exclude** to exclude playbook logs with the specified tags. You will observe that the `#system` tag is already added as a tag in the Exclude mode, which means the any playbook with the `system` tag will be excluded from the playbook logs. To include only those playbook logs with the specified tags, click **Only Include**. For example, if you only want to view the logs of phishing playbooks, i.e., logs of playbooks that have `phishing` tag, click **Only Include** and type `phishing` in the **Tags** field. You must also remove the **system** tag from the **Only Include** mode, since otherwise playbook logs with both the `phishing` and `system` tags will be included.

> You can specify a comma-separated list to Include all tags or Exclude all tags. You cannot have a mix of Include and Exclude tags.

Filters will apply from the time you have set the filter, i.e., if you have added a `phishing` tag in the **Exclude** list at 16/05/2019 17:00 hours, then the filter will apply only from this time. The historical logs, logs before 16/05/2019 17:00 hours will continue to display in the Executed Playbooks Logs.

From version 7.0.1 onwards, the settings of the playbook execution history logs that are filtered using tags have been updated as follows:

- The 'global' filter will be applicable on only on the global execution log list.
- Record level playbook execution history log will show all playbook logs by default; users can apply temporary filters to filter the result. The **Set as default filter for all users** option has been removed from the record level playbook execution log filter settings.
- Playbook-level playbook execution log will show all logs. When you open a playbook in the playbook designer and view its execution history, you will see all the logs. There is no UI option in playbook execution history to filter the logs. For example, the default exclude filter that is applied for '`system`' playbooks will not be applicable when you open the playbook execution log for a specific playbook by clicking **Tools** > **Execution History** in the playbook designer.

An example of excluding playbook logs by tag follows:
If you have added tags such as `dataIngestion` in your playbooks, then you can filter out the data ingestion logs by clicking **Exclude** and typing `dataIngestion` in the **Tags** field. If an administrator with `Update` rights on the `Security` module wants this filter to be visible to all users, then the administrator can save this filter as a default for all users, by clicking the **Set as default filter for all users** checkbox and then clicking the **Save & Apply Filter** button.

From version 7.0.1, the **Set as default filter for all users** checkbox has been removed from the record level playbook execution log filter settings.

If you do not have appropriate rights, you can apply the filter for only yourself by clicking the **Apply Filter** button and view the filtered playbook executed logs.
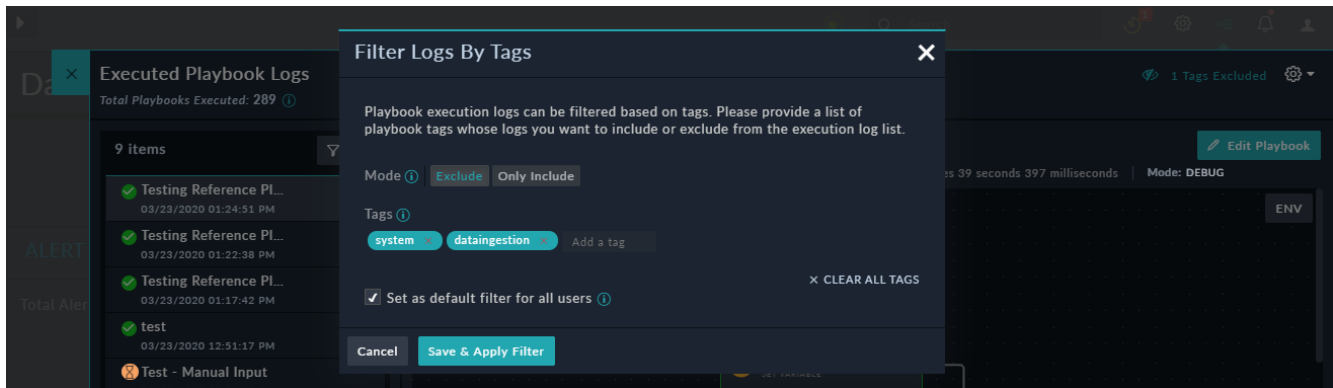
This applies the filter and displays text such as **2 Tags Excluded** on the top-right corner of the Executed Playbook Logs dialog. Now, the Executed Playbook Logs will not display logs for any *system* playbook or for any *data ingestion* playbook.
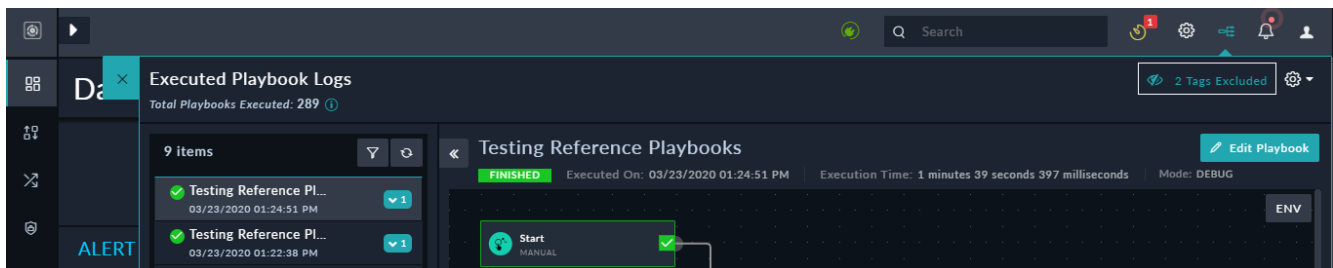
Users (without administrative rights) can remove filters by clicking **Settings** > **Filter Logs by Tags** or clicking the `<number of Tags included>` link to display the `Filter Logs By Tags` dialog and click **Clear All Tags** to remove the tags added and add their own tags. However, these changes will only be applicable till that instance of the log window is open. If the page refreshes or the window reloads, then the tags specified by the administrator will again be applied.

Terminating playbooks

You can terminate playbooks that are in the **Active**, **Incipient**, or **Awaiting** state. Users who have `Read` and `Execute` permissions on the `Playbooks` module can terminate a running instance of their own playbook instance. Administrators who have `Read` permissions on the `Security` module and `Execute` permissions on the `Playbooks` module can terminate running instances of any playbook.

To terminate a running playbook instance, open the Executed Playbook Logs and click the instance that you want to terminate and click **Terminate** as shown in the following image:
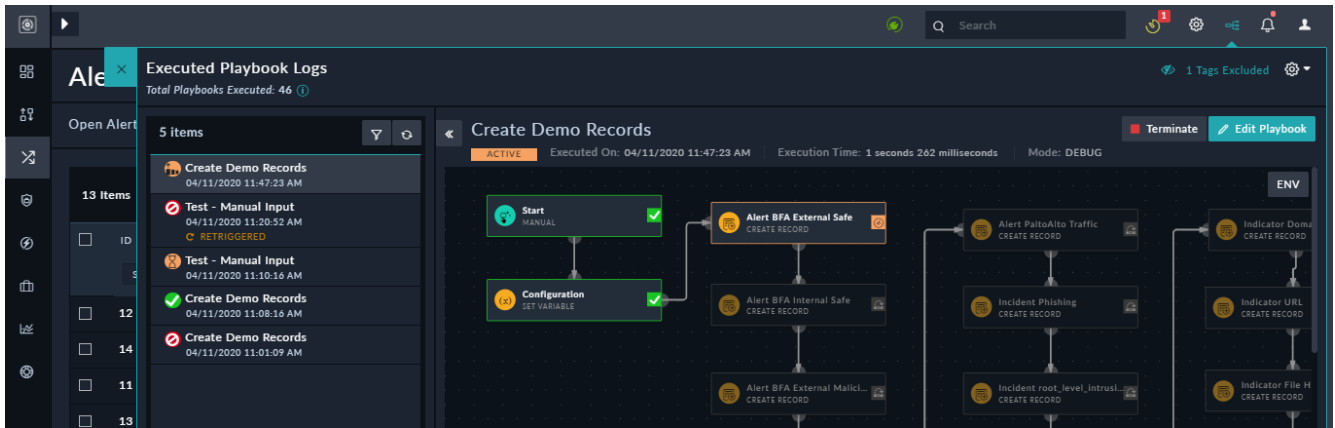
Once you click **Terminate**, the `Terminate Execution` dialog is displayed in which you can choose to either terminate only the particular running instance, by clicking **Terminate Current Instance Only** or terminate all running instances, by clicking **Terminate All Running Instances**.
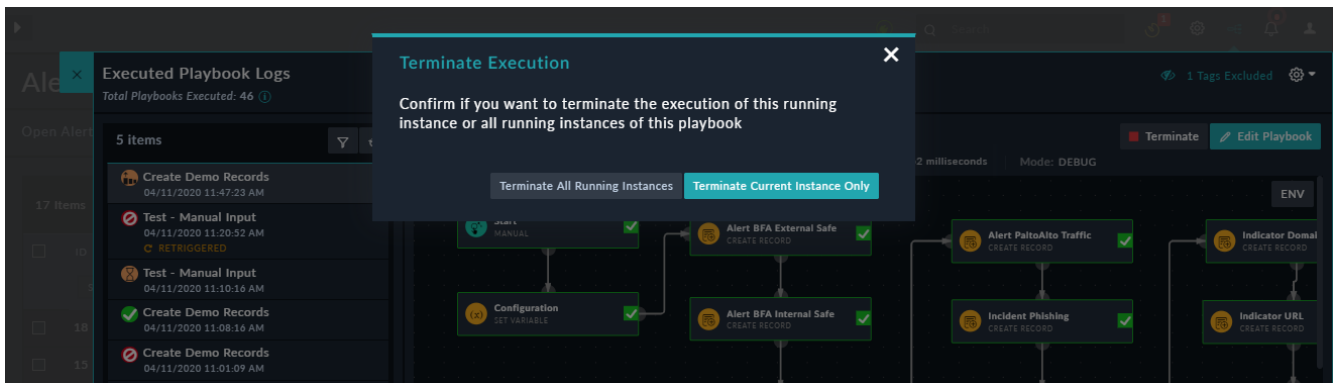

If you click **Terminate Running Instance Only**, then the state of that playbook changes to **Terminated**:
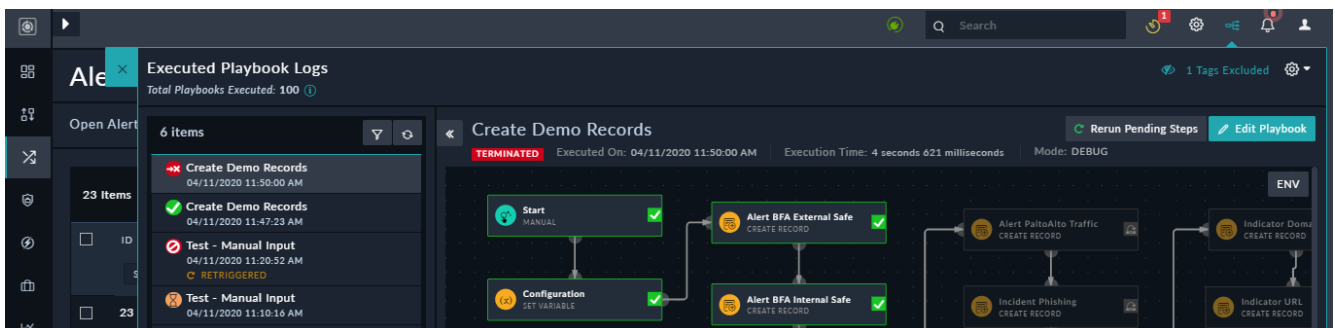
You can also choose to terminate the running instances of all playbooks that are in the Active, Incipient, or Awaiting state.

To terminate the running instances of all playbooks based on the status of the playbooks, do the following:

1. Click the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog.
2. Select the **Terminate Running Instances** option, which displays the `Terminate Running Instances` dialog.
3. In the `Terminate Running Instances` dialog, select the status (**Active**, **Incipient**, or **Awaiting**) whose running instances of Playbooks you want to terminate, and click **Terminate**.
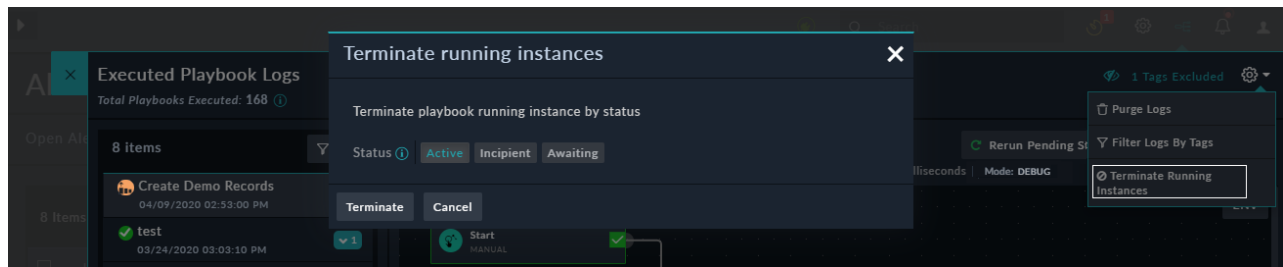
You can rerun the playbook from the step it was terminated by clicking the **Rerun Pending Steps** button on the terminated playbook.

Setting up auto-cleanup of workflow execution history

Workflow Execution history is extensively persisted in the database for debugging and validating the input and output of playbooks at each step. A very large execution history, however, causes overhead regarding consumption of extra disk space, increase in the time required for upgrading FortiSOAR, etc. Therefore, it highly recommended to set up an auto-cleanup of the workflow execution history using a weekly cron schedule.

To delete the workflow run history keeping the last 'X' entries, `ssh` to your FortiSOAR appliance as `root` and run the following command:

```
# /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --
keep X
```
For example, to delete all workflow run history, apart from the last 1000 entries, use the following command:
```
# /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --
keep 1000
```

To set up a weekly schedule delete workflow history, to the above command, add a cron expression entry in the `/etc/crontab` file that would schedule a workflow execution history cleanup as per your requirements. Command to edit a cron job is `crontab -e`.

For example, the command to add an entry in the `/etc/crontab` file that would schedule a workflow execution history cleanup to every Saturday night and delete all workflow run history, apart from the last 1000 entries, would be as follows:
```
# 0 0 * * SAT /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py
cleandb --keep 1000
```

Note that running the above command deletes the workflow entries but does not release the disk space back to the OS, i.e., it keeps it reserved for the Postgres process. This is the desired behavior, and no further action is required if the execution history cleanup is scheduled because the Postgres process would need the freed-up disk space to store further workflows. If, however, you also wish to reclaim disk space for backup or restore or other activities, you would additionally need to run a "full vacuum" on the database after you `ssh` to your FortiSOAR appliance as `root` and run the following commands:

```
psql -U cyberpgsql sealab
psql (10.3)
Type "help" for help.

sealab=# vacuum full;
VACUUM
sealab=# \q
```

Known Issue: If you do not schedule the workflow execution cleanup, and you are deleting a very large set of entries in one go, then the `db cleanup` command might fail due to the failure in loading the large set of entries into memory. In

this case, you will have to run the command in batches.

For example:

```
# /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --
keep 100000
# /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --
keep 90000
```

Disabling Playbook Priority

You can disable playbook priority, in case playbook priority queuing is hampering your system's performance. Version 7.0.0 onwards, FortiSOAR uses rabbitmq as the message broker, and because of this priorities of an already declared queue in rabbitmq cannot be changed dynamically; for more information, see https://www.rabbitmq.com/priority.html#using-policies.

Therefore, from version 7.0.0 onwards, use the `ENABLE_PRIORITY` setting in the workflow engine to enable/disable playbook priority. However, before changing the `ENABLE_PRIORITY` setting, first delete the `celery` queue, after you have ensured that no data is present in the `celery` queue.

To enable/disable the `ENABLE_PRIORITY` setting in `/opt/cyops-workflow/sealab/config.ini`, do the following:

1. List queues and check the `celery` queue and its count of messages, use the following command:
    ```
    rabbitmqctl list_queues -p intra-cyops
    ```
    If the `celery` queue has zero messages in it, then the output would be:
    ```
    celery 0
    ```
    This ensures that no data is present in the `celery` queue.
2. Delete the `celery` queue , using the following command:
    ```
    rabbitmqctl delete_queue celery -p intra-cyops
    ```
3. Change the `ENABLE_PRIORITY` flag to `false` to disable playbook priority, or `true` to enable playbook priority.
4. Restart `celeryd` using the `# systemctl restart celeryd` command.

Optimizing Playbooks

Playbook steps that were looped (using the `Loop` option in the playbook step) can be run either in a sequentially or in parallel. For more information, see the `Loop` topic in the Triggers & Step chapter.

You can tune the thread pool size and other settings for parallel execution using the settings that are mentioned in the following table:

Key name and location	Description	Default value

`THREAD_POOL_WORKER` `/opt/cyops-workflow/sealab/sealab/config.ini`	The thread pool size is used for parallel execution. The `THREAD_POOL_WORKER` variable is used to optimize the parallel execution and enhance performance. You can reduce the default value of the thread pool size from the default value if: 1. The number of cores on your FortiSOAR instance are lesser than the default recommended. 2. The task to be executed in the loop step is synchronous in nature and thread context switching would be an overhead.	8
`SYNC_DELAY_LIMIT` `/opt/cyops-workflow/sealab/sealab/config.ini`	If the delay specified in the playbook step is higher than this threshold, then the loop step will be decoupled from the main playbook and run asynchronously. For example if you set the `SYNC_DELAY_LIMIT` to **60**, it means that a 60 seconds check is added, and after 60 seconds the playbooks should run in parallel. This works in parallel with your playbook soft limit time, `CELERYD_TASK_SOFT_TIME_LIMIT` parameter. The time set in the `CELERYD_TASK_SOFT_TIME_LIMIT` parameter must be greater than the time set in the `SYNC_DELAY_LIMIT` parameter.	60
`CELERYD_TASK_SOFT_TIME_LIMIT` `/opt/cyops-workflow/sealab/sealab/config.ini`	To change the soft time limit for playbooks. The soft time limit value is set in seconds.	1800
`CELERYD_TASK_TIME_LIMIT` `/opt/cyops-workflow/sealab/sealab/config.ini`	To change the time limit for playbooks. The time limit value is set in seconds. **Note**: This value should always be higher than the SOFT_TIME_LIMIT. For more details, see the Celery 4.3.0 documentation >> User guide: task_soft_limit section.	2400
`CELERYD_OPTS` `/etc/celery/celeryd.conf`	To optimize the parallel running of threads in celery so that your overall playbook execution time is reduced. By default, the workflow engine spawns a separate process running a workflow. If the tasks in the workflow are asynchronous and short lived, the thread-based workers can be enabled. For more details, see the Celery 4.3.0 documentation > User guide > Concurrency >> Concurrency with Event section.	`CELERYD_OPTS="-P=eventlet -c=30"`

These optimizations also help in scaling your playbooks by resolving bottleneck that slow down playbook execution and resolving internal timeout issues.

FortiSOAR supports parallel branch execution of playbooks. Parallel branch execution optimizes playbook execution by having the ability to execute two or more independent paths parallelly.

You can enable or disable parallel execution by changing the value (true/false) of the `PARALLEL_PATH` variable in the `[Application]` section in the `/opt/cyops-workflow/sealab/sealab/config.ini` file. By default, a fresh install of version 5.1.0 will have the `PARALLEL_PATH` variable set as `true`.

Troubleshooting Playbooks

Filters in running playbooks do not work after you upgrade your system in case of pre-upgrade log records

You can apply filters on running playbooks using the **Executed Playbook Logs**. These filters will apply to log records that are created post-upgrade and will not apply to log records that were created pre-upgrade.

For log records that were created before the upgrade, use the playbook detail API:
`GET: https://<FortiSOAR_HOSTNAME/IP>/api/wf/api/workflows/<playbook id>/?format=json`

To get the playbook id, use the playbook list API:
`GET: https://<FortiSOAR_HOSTNAME/IP>/api/wf/api/workflows/?depth=2&limit=30&ordering=-modified`

Playbooks are failing, or you are getting a No Permission error

Resolution

When the `Playbook` does not have appropriate permissions, then playbooks fail. `Playbook` is the default appliance in FortiSOAR that gets included in a new team.

If you cannot access records, such as alerts, then you must ensure that you are part of the team or part of a sibling or a child team that can access the records, and you must have appropriate permissions on that module whose records you require to access or update. Only users with `CRUD` access to the `Appliances` module can update the `Playbook` assignment. For more information on teams and roles, see the *Security Management* chapter in the "Administration Guide."

Playbook fails after the ingestion is triggered

There are many reasons for a playbook failure, for example, if a required field is `null` in the target module record, or there are problems with the Playbook Appliance keys.

Resolution

Investigate the reason for failure using the **Playbook Execution History** tab (earlier known as **Running Playbooks**) in the `Playbook Administration` page. Review the step in which the failure is being generated and the result of the step, which should contain an appropriate error message with details. Once you have identified the error, and if you cannot troubleshoot the error, contact the FortiSOAR support team for further assistance using the Fortinet Customer Service & Support web portal at https://support.fortinet.com/.

Incorrect Hostname being displayed in links contained in emails sent by System Playbooks

When you are using a system playbook that sends an email, for example, when an alert is escalated to an incident, and an Incident Lead is assigned, then the system playbook sends an email to the Incident Lead specified. The email that is sent to the Incident Lead contains the link to the incident using the default hostname.

Resolution

To ensure that the correct hostname is displayed in the email, you must update the appropriate hostname as per your FortiSOAR instance, in the Playbook Designer as follows:

1. Open the Playbook Designer.
2. Click **Tools** > **Global Variables** to display a list of global variables.
3. Click the `Edit` icon in the `Server_fqhn` global variables, and in the `Field Value` field add the appropriate hostname value.
4. Click **Submit**.
   The system playbook will now send emails containing the updated hostname link.

> In the system playbook (or any playbook) that is sending an email, ensure that you have used the `Server_fqhn` global variable in the `Send Email` step.

Purging executed playbook logs issues

If you are facing issues while purging of executed playbook logs such as, the purge activity is taking a long time or the purging activity seems to be halted, then you could check if the `Soft time limit (600s) exceeded for workflow.task.clean_workflow_task[<taskid>]` error is present in the `/var/log/cyops/cyops-workflow/celeryd.log` file. The `Soft time limit` error might occur if the amount of playbook logs to be purged is very large.

Resolution

Increase the value set for the `LOG_PURGE_CHUNK_SIZE` parameter, which is present in the `[application]` section, of the `/opt/cyops-workflow/sealab/sealab/config.ini` file.

By default, the `LOG_PURGE_CHUNK_SIZE` parameter is set to `1000`.

Playbooks fails with the "Too many connections to database" error when using the "parallel" option for a loop step in Playbooks

Playbooks can fail with the `Too many connections to database` error when you have selected **Parallel** in a loop step to execute playbook steps in parallel.

Resolution

To resolve this issue, reduce the number of parallel threads. To reduce the number of parallel threads, you have to change the value of the `THREAD_POOL_WORKER` variable. The `THREAD_POOL_WORKER` variable is present in the `/opt/cyops-workflow/sealab/sealab/config.ini` file, and by default the value of this variable is set to **8**.

Frequently Asked Questions

Q: Is there a way to force variables set in a reference playbook to carry over into the parent playbook? I rather not put a group of steps I need in the parent if I can avoid it, as I am using the child playbook as an action itself, so would it duplicate the functions?

A: In general, variables set in child playbooks do not carry over to the parent playbook. The one exception is that the **Reference a Playbook** step will return (in `vars.result`) the return value of the last executed step in the child playbook. For instance, if the last step in the child playbook is **Find Record**, then the **Reference a Playbook** step will populate `vars.result` with the records that have been found using the **Find Record** step.

If u want to define the playbooks result as a combination of results of previous steps or sub-steps, you can use the `Set Variable` step at the end of the playbook and define variables that would contain data that you require to be returned.

Q: How do I convert Epoch time returned by a SIEM to a datetime format?

A: If you have a playbook, which has a connector step that connects to a SIEM, such as ArcSight or QRadar, and the SIEM returns the result in Epoch time (milliseconds), then you can convert Epoch time to the datetime format using the following command:

```
# arrow.get(1531937147932/1000).to('Required Timezone').strftime("%Y-%m-%d %H:%M:%S %Z%z")
```

or

```
# arrow.get(1531937147932/1000).to('Required').format('YYYY-MM-DD HH:mm:ss ZZ')
```

For example,

```
# arrow.get(1531937147932/1000).to('EST').format('YYYY-MM-DD HH:mm:ss ZZ')
```

Will return the following output:

```
2018-07-18 14:05:47 EDT-0400
```

For more examples on dates and times used in Python, see http://arrow.readthedocs.io/en/latest/.

Q: How do I change the timeout limit for playbooks?

A: To change the time limit or soft time limit for playbooks you must edit the `CELERYD_TASK_TIME_LIMIT` and `CELERYD_TASK_SOFT_TIME_LIMIT` parameters in the `/opt/cyops-workflow/sealab/sealab/config.ini` file. By default, these parameters are set in seconds, as follows:

```
CELERYD_TASK_TIME_LIMIT = 2400
CELERYD_TASK_SOFT_TIME_LIMIT = 1800
```

Once you have made the change you must restart all the FortiSOAR services by using `csadm` and running the following command as a *root* user: :

```
# csadm services --restart
```

Tutorial: Creating a Sample Playbook to determine maliciousness of an indicator in FortiSOAR

Purpose

This tutorial aims to walk you through the steps you require to create a simple playbook in FortiSOAR. This playbook aims to determine whether a specific indicator (of type "Domain") is malicious or not.

This tutorial aims to provide you with examples of how to use Playbook Triggers and Steps, Connectors, and Dynamic Values that have been explained previously in this guide.

Steps to create the sample playbook

1. Log on to FortiSOAR using your credentials.
2. Click **Automation** > **Playbooks**.
3. On the **Playbook Collections** tab, click **+ New Collection** and in the **Add New Playbook Collection** dialog, add a name for the new playbook collection. For our example, type the name of the playbook collection as **Test**.
4. In the **Test** playbook collection, click **Add Playbook** to add a new playbook and in the `Add New Playbook` dialog, add a name for the new playbook. For our example, type the name of the playbook `Determine whether Domain is Malicious` and click **Create**.
   This displays the **Playbook Designer**.
5. In the Playbook Designer, select the **On Create** Trigger as the Trigger step, since we want to run this playbook once an Indicator of type `Domain` is added in FortiSOAR. The Step Name can be retained as **Start**. From the **Resource** drop-down list, select **Indicators**. In the **Trigger Condition** section, click **Add Condition.** In the Condition Builder add **Type Equals Domain**, as shown in the following image:
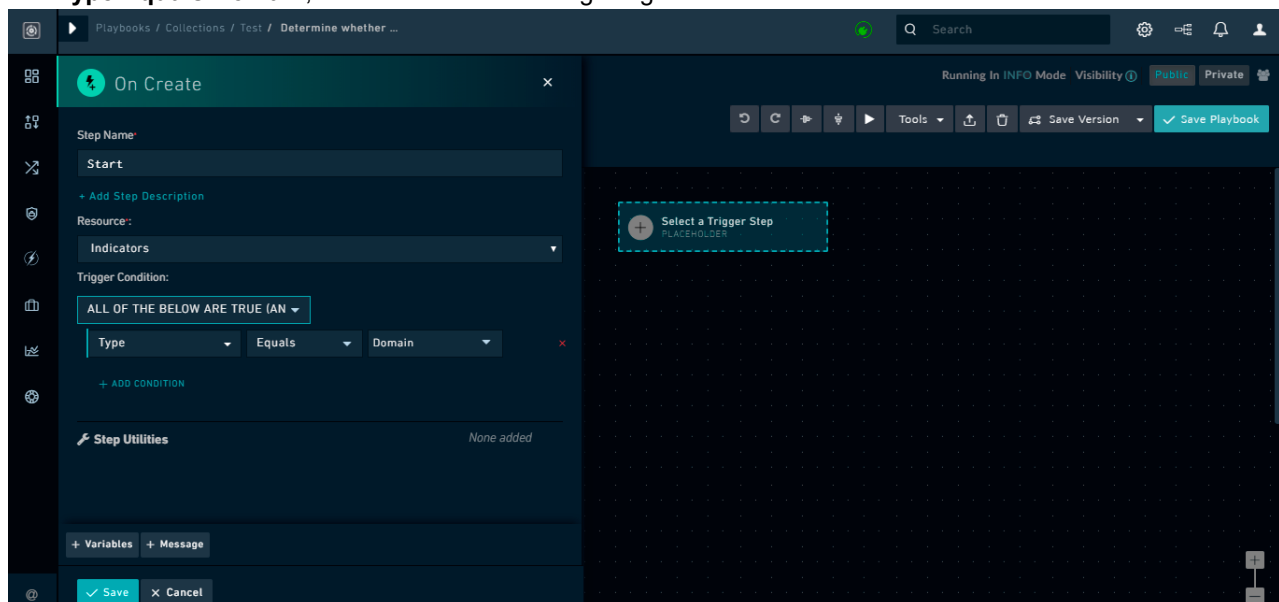
Click **Save** to save the Start step.

---

6. The next step in this playbook is to check the reputation of the domain once an Indicator record of type "Domain" is added in FortiSOAR.

For this, you can use any third-party services that analyze suspicious domains. FortiSOAR supports a number of such services or tools in the form of connectors that you can very quickly configure and directly use as a step, in playbooks, to check reputations of domains, files, IP address, URLs, etc. Add another step i.e., the **Connectors** step and join the *Connector* step to the *Start* step. For our sample, we are using the *VirusTotal* connector (assumption here is that the VirusTotal connector is configured in your system).

On the `Connectors` page, in the **Search Connectors** text box, type **VirusTotal**:
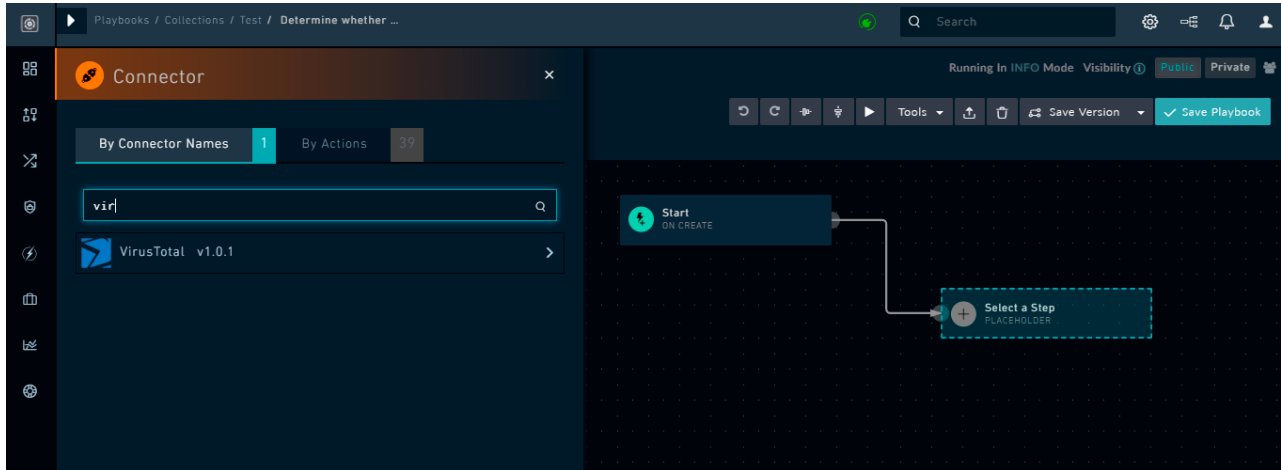
Click the VirusTotal row to add the connector step.

In the **Step Name**, type `Get Domain Reputation`.

From the **Configuration** drop-down list, select the configuration you have created for VirusTotal. From the **Action** drop-down list, select **Get Domain Reputation** since in our example we want to retrieve the reputation for a domain.

In the `Inputs` section, in **Domain** use *Dynamic Values* to populate the value of the domain you have added from the record:
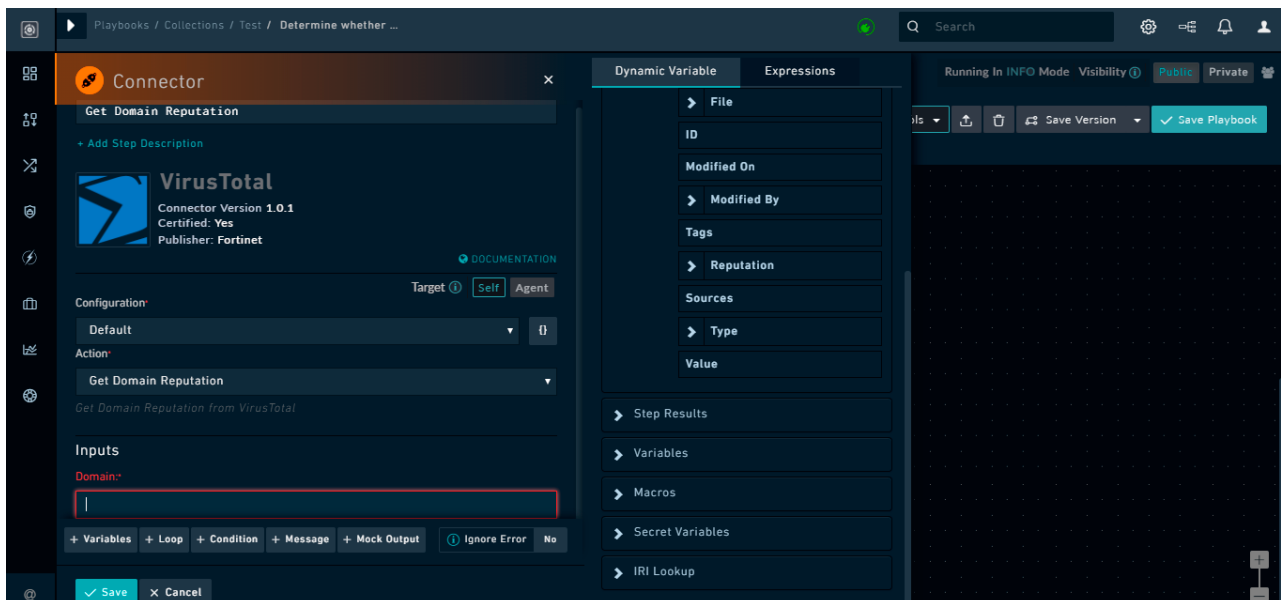
Dynamic Values will populate `{{vars.input.records[0].value}}` in the Domain field as shown in the following image:
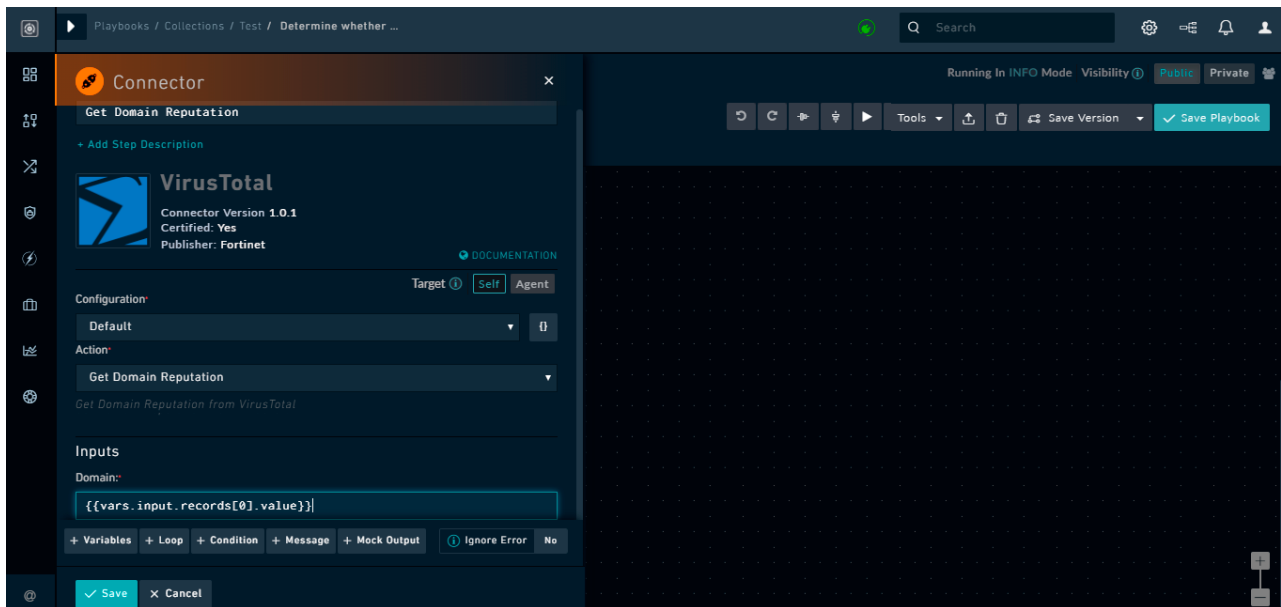
Click **Save** to save the Connectors step.

7. Add a **Decision** step to determine whether the domain is malicious or not. Currently, choose the **Decision** Step and add the step name as **Is Malicious**. We will add the conditions after we add the other steps. Click **Save** to save the Decision step.

8. Update the Record if the Domain is determined to be Malicious. Add the **Update Record** step. In the **Step Name**, type `Set as Malicious`.
   From the **Model** drop-down list, select **Indicators**.
   In **Records IRI** use Dynamic Values select the ID of the newly added record:
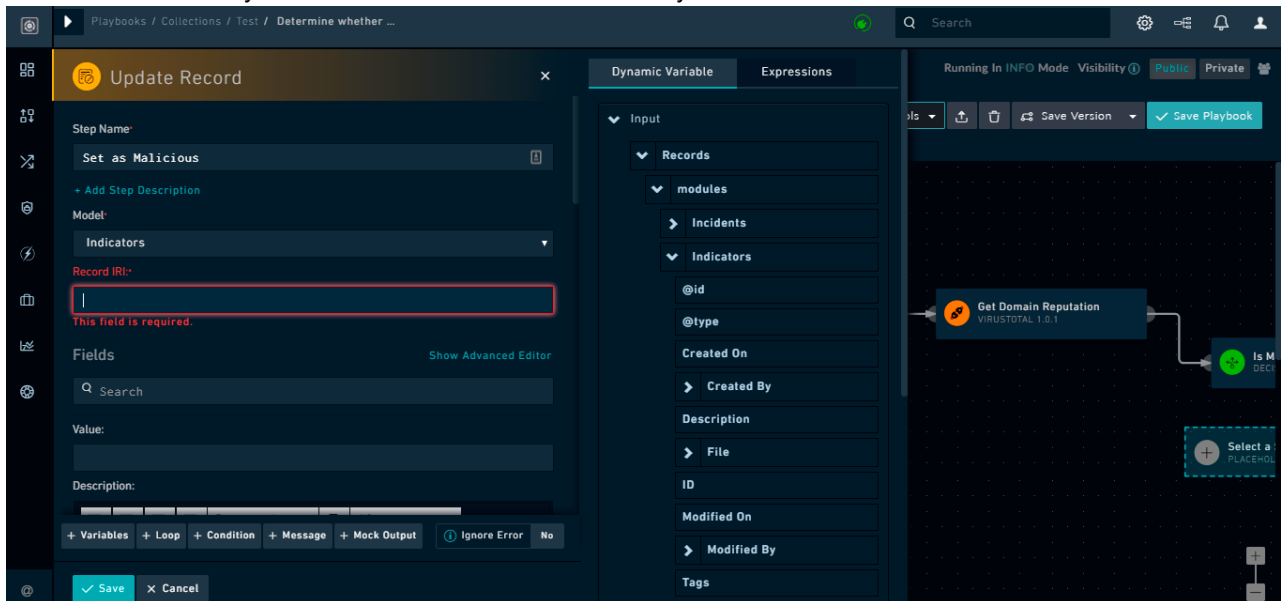
Dynamic Values will populate `{{vars.input.records[0]['@id']}}` in the Record IRI field as shown in the following image:
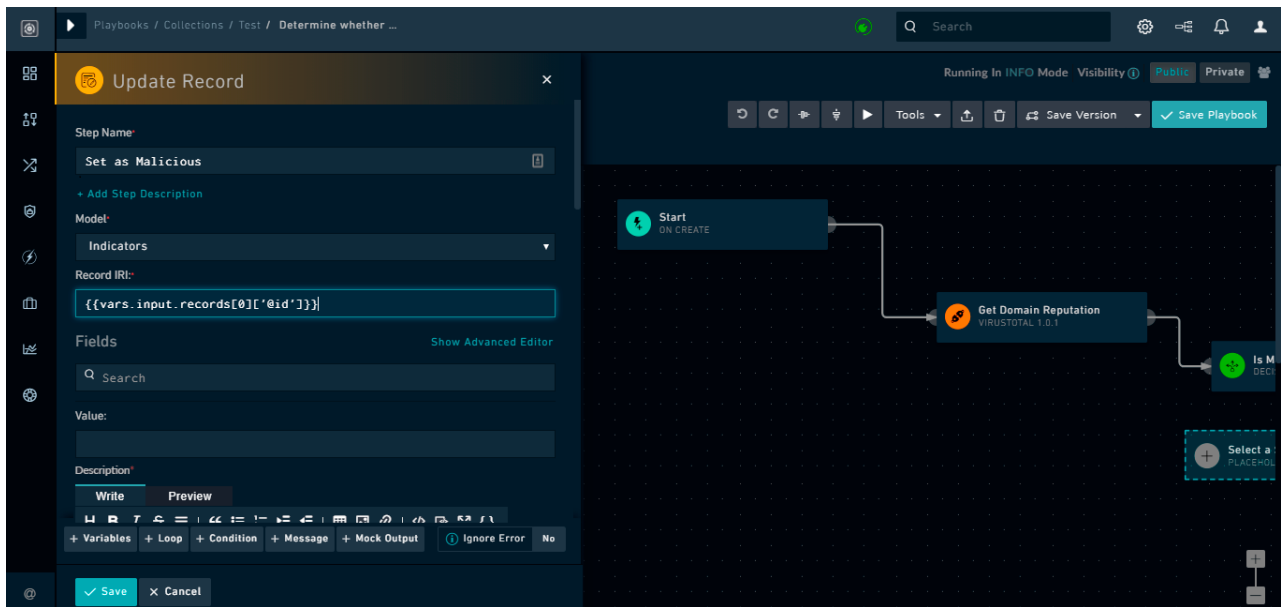
In the **Description** field, add the description that you want to see in the Indicator record. You can use formatting and Dynamic Values in the Description field.

For our example, we have added **Aggregate Information** that is gathered from VirusTotal about the domain and the **Verbose** message:
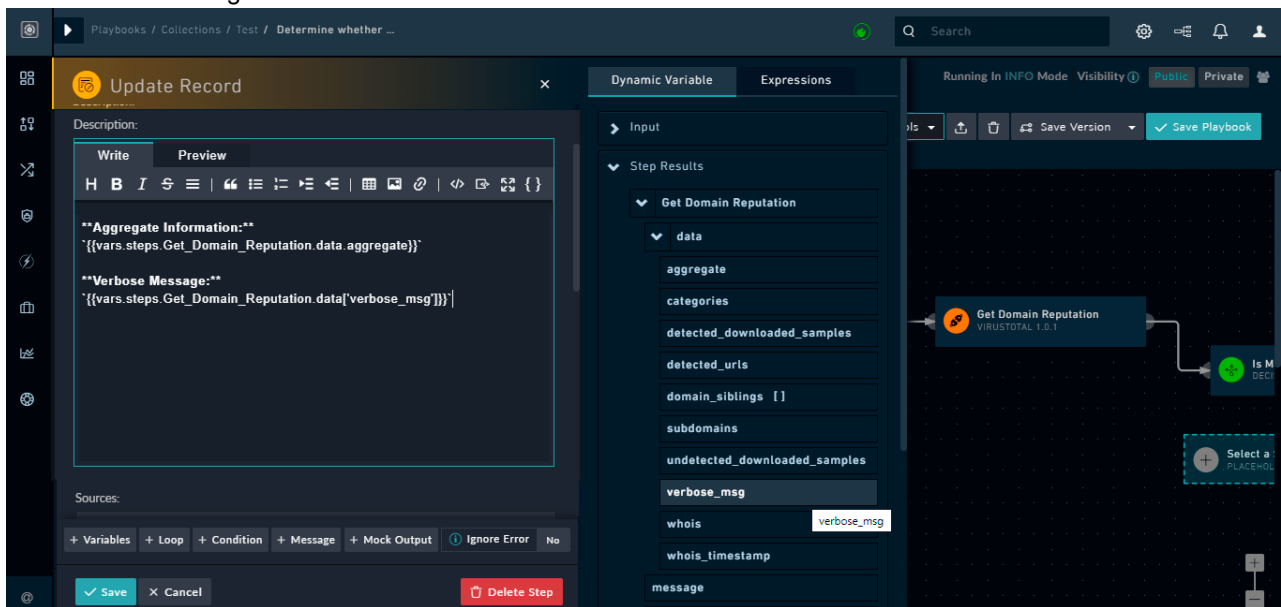
You should also select **Malicious** in the Reputation field, so that the reputation of the indicator in the record will be updated to *Malicious*.

Click **Save** to save the Update Record step.

9. Similarly, Update the Record if the Domain is determined not to be Malicious. Add the **Update Record** step.

In the **Step Name**, type `Set as Good`.

From the **Model** drop-down list, select **Indicators**.

In **Records IRI** use Dynamic Values select the ID of the newly added record or add `{{vars.input.records[0]['@id']}}`.

In the **Description** field, add the description that you want to see in the Indicator record, similar to what you have added in the **Set as Malicious** step.

You should also select **Good** in the Reputation field, so that the reputation of the indicator in the record will be

updated to *Good*.

Click **Save** to save the Update Record step.

10. Add conditions to the **Is Malicious** decision step:

To add conditions, click **Add Condition**.

To add conditions for updating the record if the domain is **Malicious**:

In the `Condition 1` section, click the **Show Advanced** link.

In the **Condition** box, use Dynamic Values to determine the domain reputation value retrieved from VirusTotal:
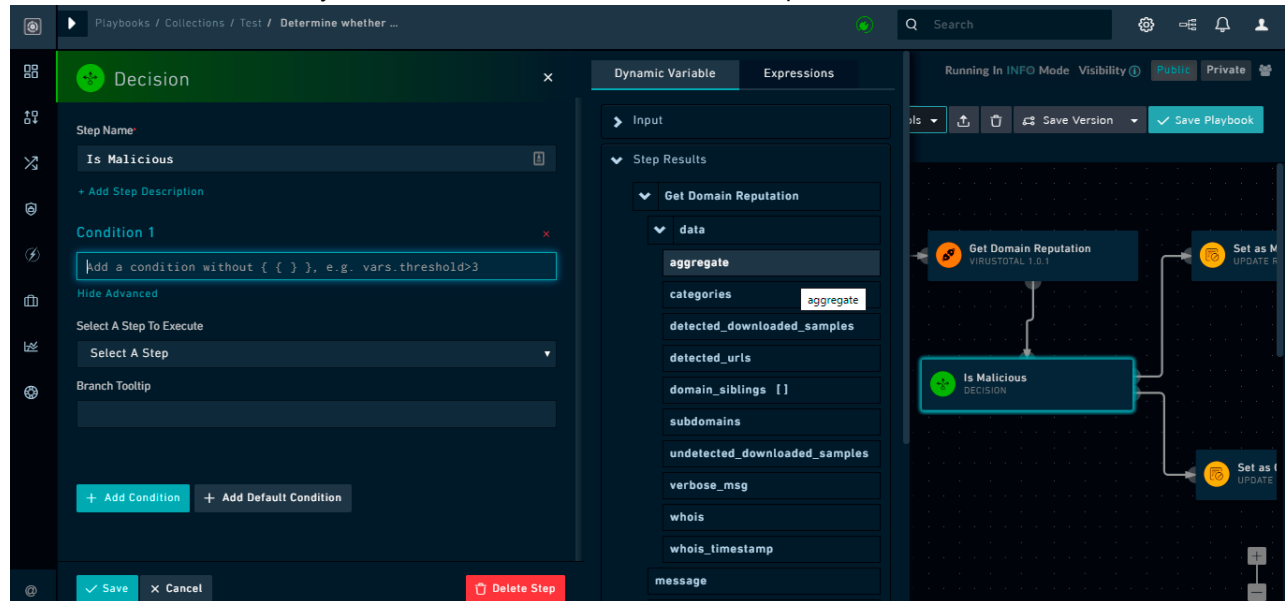
This will add the condition `vars.steps.Get_Domain_Reputation.data.aggregate`.

Now for our example, we will set a domain to Malicious if its aggregate value is greater than or equal to 5. Therefore, the condition must appear as `vars.steps.Get_Domain_Reputation.data.aggregate >= 5`.

Then, from the **Select A Step To Execute** box, select the **Set As Malicious** step. You can also provide a tooltip for the branch in the **Branch Tooltip** field, in this case, added `Yes`:
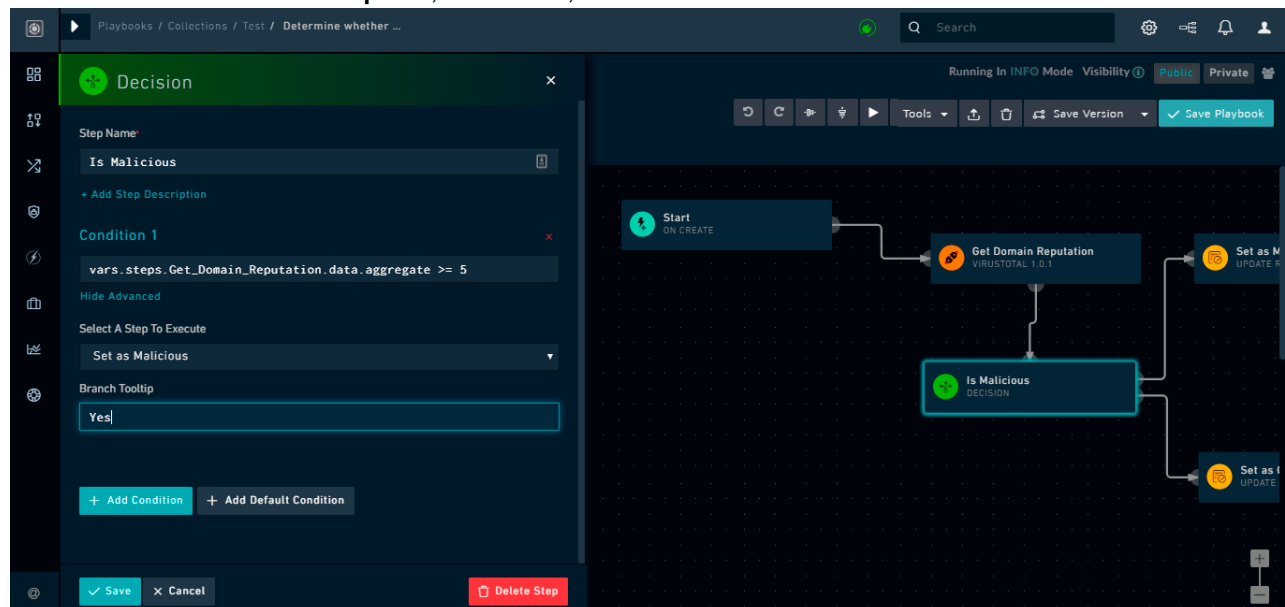
Similarly, click **Add Conditions** again to add conditions for updating the record if the domain is **Good**:

Click **Add Condition** and in the **Condition** box, add the condition `vars.steps.Get_Domain_Reputation.data.aggregate < 5` since for our example, we will set a domain to Good if its aggregate value is

lesser than 5.
Then, from the **Selected A Step To Execute** select **Set As Good**, and provide the tooltip `No` for the branch in the
Branch Tooltip field.
Click **Save** to save the Decision step.

11. Click **Save** to save the playbook and ensure that the playbook is active:
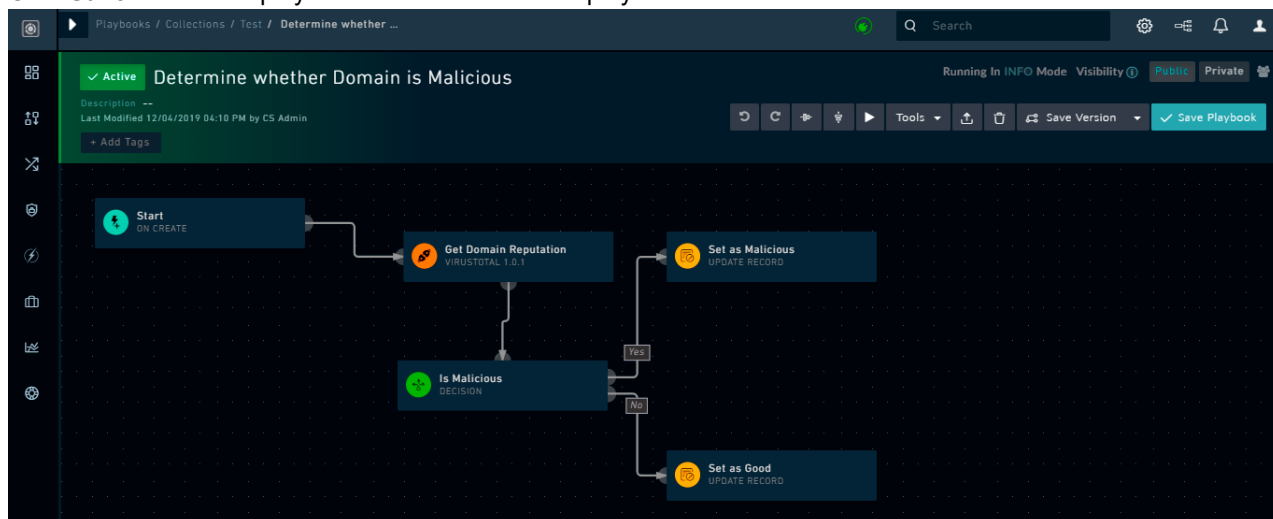
If the playbook is not active then click the **Inactive** button to activate the playbook.

12. To check if the playbook is working as expected, click **Incident Response** > **Indicators** and on the Indicators page,
click **+ Add Indicator**.

13. In the **Create New Indicator** dialog, from the **Type** drop-down list, select **Domain** and in the **Value** field, type
`SPIT.COM` (whose domain reputation is **Good**) and click **Save**.

This will trigger the playbook, and the domain reputation will be retrieved from VirusTotal as shown in the detailed
view of the record:
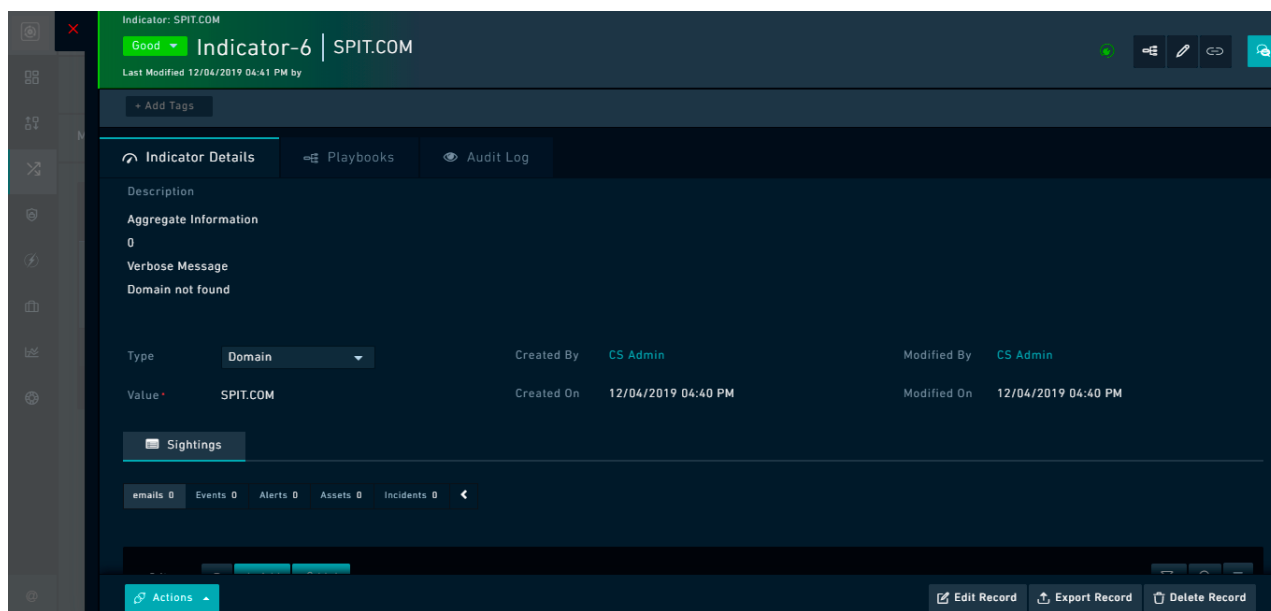
Similarly, if you add a domain whose reputation is malicious, for example, `gumblar.cn`, the playbook will again be triggered, and the domain reputation will be retrieved from VirusTotal as shown in the detailed view of the record:
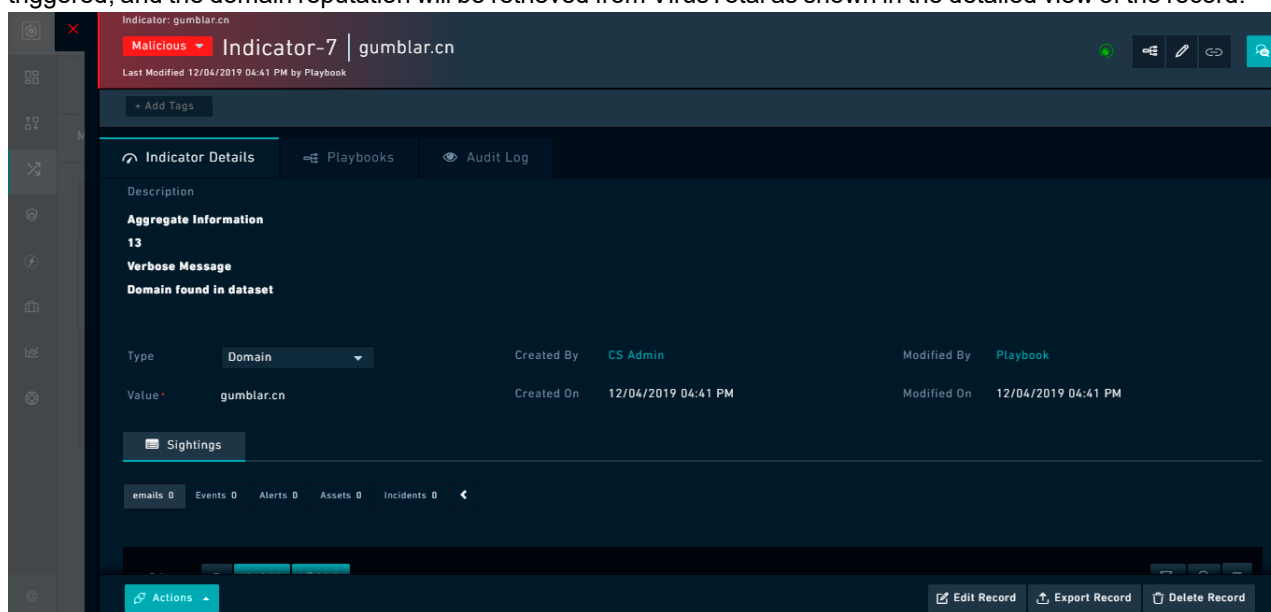
Conclusion

This tutorial demonstrates how you can create a very simple playbook to determine the reputation of a domain.

Using this as a base you can create very complicated playbooks to automate all your security investigation workflows, for example you can use the alert data that you have received from your SIEM to check the reputation of the IP address or domain directly in the alert data and if it is determined to be malicious then directly block that IP address or domain, without the need of intervention from analysts and analysts can focus on more critical aspects of investigation.
>

FORTINET