Standardization

TIP Group Space

Exported on 12/27/2022

Table of Contents

1 Error Handling

1.1 Input Parameters Error Handling

Action Type	Case	Case Handling
List actions	Action has a Max Rows to Return input parameter, zero or negative value is provided as input	Action should return error, not execute and print the following error message: “Positive number should be provided for Max Rows to Return“
List actions	In development of previous case, if we know that there is an API limit for Max Rows to Return, or we want to have a specific limit for that field, and if unsupported value is provided	Action should return error, not execute and print the following error message: “Positive number from 1 to <max value> should be provided for Max Rows to Return“
Update actions	Action needs a specific set of parameters, for example: in Update Alert action, to change the status of alert to closed, alert close reason should be provided as well. If that minimum set of input parameters is not provided, we can print an error and stop the action. Those error messages can be provided for all the cases where we know the minimum values, and those values are not provided.	Action should return error, not execute and print the following error message: “To change the status of alert to closed, please provide both new alert status (closed) and closed reason value“.
List Actions, Update Actions	Action has an input parameter that accept value in a specific format, for example IP address or Time value.	Action should return error, not execute and print the following error message: “Wrong format provided for the <input parameter>. Accepted format: <format description>“.
All Actions	Action accepts a JSON formatted text as input parameter.	if there is JSON query as a param the action should validate the query and return error in case the JSON is invalid.

2 Integration Best Practices v1

Disclaimer

This is work in progress, not final version

Intro

Actions Best Practices are aimed to provide guidance for Products, DEV and QA on Siemplify Marketplace integrations actions - action descriptions, naming, inputs/outputs and so on.
The goal of best practices is to accumulate and share good practices on action design that later can be transformed in a more formal standard that we can follow and check against to have a more mature Marketplace!

Content

General Actions Best Practices - accumulate guidelines that all actions should aim to follow.
Specific actions guidelines, for example for "List" Actions, contain guidelines applicable to this specific action type and should be considered in addition to general best practices.

Best Practices can provide recommendations on the following action elements:

- Action Naming
- Action Description
- Action Inputs
- Action Outputs - script_result, json_result
- Action Execution Flow
- Error Handling
- Case Wall
- Release Notes

2.1 General Actions Best Practices

2.1.1 Naming

1. Action Name should imply on what action does, for example: Enrich Entities, Add Comment To Incident, Isolate Machine.
2. Name should be Capitalized, linking words ("a", "to", "for" and so on) should be in lower case.
3. The less words the better.
4. If action works with Siemplify entities, its recommended to put "Entity" in the naming to imply action works on entities.

2.1.1.1 Examples of naming for reference:

- Microsoft Azure Sentinel Integration: “List Incidents” action.
- AWS EC2 Integration: “Start Instance” action.

2.1.2 Description

1. Action description should continue the explanation made with the action name. If its an enrichment, it should state in the description that this action for example is enriching the hostname and ip entities.

2. If the action is running on entities, action description should specify which entity types are supported.
   If entities are not supported - description should explicitly state that entities are not used (not mandatory, depending on the context).
3. If there are any specific notes users have to be aware when running the action - action description should include such notes, for example, it can be in a form:

   ```
   Note: <and note description next>
   ```
   Example of such notes include: if action async or not, API rate limiting, other limitations and so on.
4. If action has requirements for expected formats for entities, for example, we expect email as the User entity, action should define it in the description, eg:

   ```
   … Supported entity types: IP Address, Username (username and/or username@domain
    formats),
   URL  (action will strip domain part of URL entity).
   ```
5. If action description reference a name of the action or action input parameters, those should by in double quotes.
6. Optionally, if the action is complex, link to the action documentation should be provided for user to check. Example:

   ```
    …For more info please refer to <link> .
   ```

2.1.2.1 Examples of descriptions for reference:

- *Simpler description:* Microsoft Azure Sentinel Integration: “List Incidents” action:

  ```
  List Microsoft Azure Sentinel Incidents based on the provided search criteria.
  ```
- *More complex description:* AWS EC2 Integration: “Start Instance” action:

  ```
  Starts an Amazon EBS-backed instance that you have previously stopped.
  It can take a few minutes for the instance to enter the running state.
  Notice that you can't start an instance store-backed instance.
  For more information about instance store-backed instances,
  see Storage for the root device.
  ```

2.1.3 Action Inputs

1. Action input description should start with “Specify…” for all non-boolean parameters.
2. For boolean parameters, input description should start with “If enabled…”.
   Action input parameter names should be Capitalized, linking words ("a", "to", "for" and so on) should be in lower case.
3. Special characters shouldn’t be a part of the input parameter names.
4. If the parameter can only work with certain values, it is preferable to make it as a drop-down input type. If there are multiple values that can be used, then it depends on the amount of values: if there are 2-3 values, then you can create 2-3 boolean parameters that will do the job, if there are more than that, then it makes

more sense to have a string parameter that accepts multiple values. Note: description in the last case should include all possible values.

2.1.4 Action Outputs

2.1.4.1 script results

1. **is_success**. - For script result **is_success** variable should be used. **is_success** will indicate if action was executed successfully or not, this parameter will Siemplify User work with when running actions/building playbooks. If **is_success**=True action executed successfully, if **is_success**=False, action failed at some point. Exact conditions of when **is_success** should return True and when False should be clearly defined in Spec, implemented by Dev, checked by QA.
2. If action is working on entities, and it was executed on multiple entities, action should be specifically address if **is_success** should be True if at least one entity ran successfully, or if all entities ran Successfully, that logic depend on the action type and will be covered more below. Example of first case is an “Enrich Entities“ action, if we enriched at least one entity **is_success** is true, example of the other case is an “Isolate Host“ action, in which due to action importance in alert remediation, we should return **is_success** ONLY if action ran successfully on all of the provided entities.

2.1.4.2 json results

1. JSON result provide a rich information about action results to Analyst, because of it, action should return JSON result whenever possible.
2. Continuing previous thought, JSON result can provide useful insight for Analyst in case if action failed. Action should provide a JSON result in case of error whenever possible, content of json result can be custom and provide information we want to show to Analyst in case of action fail.
3. Other thing to implement (if possible): in case action was executed on multiple entities and failed, in JSON result shown it can be presented on which entities action succeeded and on which failed. It will be beneficial for Analyst' review of action results to have this data.
4. If possible, if action returns no results (empty list) we should consider adding this result as JSON result, for example we can create in JSON a new key "events" and set it to empty list ("[ ]"). This will help Analyst to define playbooks that will process this JSON result.
5. Unless specifically required, we should not change the values returned in the JSON Result from the API. We can change the returned data in JSON result in cases for example when JSON contain spaces or dots (we cant use such keys in playbook placeholders).
6. Additionally on keys returned in JSON result: keys in Json result should not have spaces or dots in it, "Key Name" is not a correct key and should not be used in action, because Analyst cant use key like this in the Expression Builder. If keys from API have spaces, it should be changed in json result to"Key_Name".
7. Keys in json results SHOULD NOT contain dynamic values (keys that can change between different action results) because it will make keys unusable with the placeholders in playbook (keys will be different every time, so we cant set them as placeholders) and Analyst will not be able to use this action in the playbook.
8. JSON result returned by action should be compatible with the Siemplify playbook expression builder.

2.1.5 Action Execution Flow

1. Action should have defined its status and result, and there is a difference between those two concepts. Action results are script results and json results described in a section above. Action status however should describe in which cases action should return success and continue playbook execution, and in which action should return error out and stop playbook execution.

2. Previously, we considered that action should not fail **unless serious** error was thrown. We considered that playbook need to continue running, and we will set action's script result, is_success to False, to indicate that the action has failed. But currently we are stepping away from this approach, and now where it makes sense, action can return error (additionally setting is_success to False). For example, if action is “Update Incident”, and action cant find provided incident id, action can fail with actual error (and is_success=False in addition) saying that the action failed to run because incident with provided incident id was not found. Positive thing from this approach is that now it is easier to spot where action failed (action will throw an actual error in addition to the is_success=False), which makes it more convenient to create, debug and run playbooks. In addition, in latest Platform versions there is an option to overwrite playbook execution in every action - playbook can skip a failed step (action) and continue execution.

2.1.6 Error Handling

1. Error Handling is a very important part of action definition. Unfortunately, error handling is case based, and we cant put all possible examples of error handling in this document. Please refer to the following document for the detailed information on error handling in actions - Error Handling(see page 5)
2. In general, action need to make sure to provided an error if minimum needed input parameters are not specified - there could be cases that action has multiple non mandatory input parameters, but expects a certain combination of those parameters to be provided. If its not the case - action should return is_success=False and error out the action.

2.1.7 Case Wall

1. Case Wall is a way for us to communicate action results to the Analyst, we need to be able to provide all of the needed information from action results. First thing that Analyst will see for the action results is an **output message** -text that should “tell“ a high level status of the action, whether it was successful or not. Text of the message should be clear, not contain grammar issues.
2. If action was working on specific entity, or object that was defined in action input param, for example incident number, we need to print in the case wall output message its name so it will be clearer for Analyst to check results..
3. If possible, action should have custom messages depending on the action results where it can be done. For example, if action=update incident and wrong incident id was provided, action should print the message indicating that wrong id was provided, and not a general error (failed to execute action for example). Other situation could be that system dont allow to update closed incidents, and user tries to update a closed one, we can do a specific error message for this case since we know about it. In other situations action can result "general" error (failed to execute action).
4. Case Wall should not have json viewer element, since it overlaps with json result. If during integration refactor an action that has a json viewer will be discovered it should be removed as excessive since we have json result.
5. If action should return a table for entity on the Case Wall, its preferred to use entity report type, which has better formatting to present entity data.

2.1.8 Release Notes

1. Release Notes are Customer facing documentation that should be aligned and follow a standard to look professional. For the guidance on how to write release notes please use the following document - Release Note - TIP Product Team[1]

[1] https://siemplify.atlassian.net/wiki/spaces/TGS/pages/3210772491/Release+Note+-+TIP+Product+Team

2.1.9 Recommended Integrations to check for reference:

VMware Lastline - https://docs.google.com/document/d/17n_u85SV3idCsHZnI-o7JKs6MSo6Ml8p57QNei505m0/edit
Microsoft 365 Defender - https://docs.google.com/document/d/1Cw_TIRIMaTJpZqIfJplkhv9gEbKQIzV7MFyPxk6g2IE/edit
Rapid 7 Insight IDR - https://docs.google.com/document/d/130gXDKVwkfeYOUcjVVKi3uvGQIW6MrzI1gou057jiT8/edit

2.2 "List" Actions Best Practices

2.2.1 Naming

1. It is recommended to keep action name short and clear, for that the following format should be used:

```
List <objects>
```

where <*objects*> could be rules, machines, policies and so on.

2.2.2 Description

1. If action will work on entities, it should state it in the description and specify which Siemplify entity types action supports.
2. If action doesnt have a DDL with possible keys for filter (see point 4 of Action Inputs), action should state in the description on which key from the response data it filters on.

2.2.3 Action Inputs

1. List action should always have a limitation of how many elements should be returned. If API doesn't support this limitation, limit should be implemented on integration (SIemplify) side.
2. If it can foreseen which values would be invalid for the action input parameters, action should return an error (fail the action) **before executing the action** in case such values were provided to the action. Example of that is a negative number for max rows to return.
3. List actions should always have a filter based on name/title. That filter should contain two logics: Exact and Contains. Use Case: finding the exact item from the whole list. Unfortunately, we can have a situation, where there are 1000 items. If user provides 1000 in Max Items to return, it will not be shown in the UI. So, the user is stuck without a solution. This filter can be optimized with API of the product, if that functionality is available.
4. To further expand on the previous point, action can have an additional drop down list (DDL) input parameter for the chosen list of keys Analyst should filter data on.
5. If possible/applicable, action should always have a parameter "Start from Row" for user to be able to specify from where to start, so in combination with "max rows to return" user will be able to loop through all of the data.

2.2.4 Action Outputs

1. Action should return a Case Wall table that will present the most important elements of the returned list.

2.2.5 Action Execution Flow

1. Action should return is_success=True if action run successfully and returned at least one result. If action run successfully, but didn't returned any results - is_success **should be false**.

2.2.6 Recommended Integrations to use for reference:

Google Cloud Compute - https://docs.google.com/document/d/1tE2HPXht0bOeuMGFUzJyqoBz8SqWZLaD3MSxJFbM1DI/edit

2.3 "Enrich Entities" Actions Best Practices

2.3.1 Naming

1. If action whole purpose is enrichment, action name should be "Enrich Entities".

2.3.2 Description

1. Description should specifically list entities types (IP, User, Hostname) action supports for enrichment.

2.3.3 Action Inputs

1. Action should have an input parameter that will create an optional insight (user can disable it) based on enrichment data. For example, if entity is marked as malicious in the target system, we can use this info for insight.

2.3.4 Action Outputs

1. Action should return enrichment information for the entities in json result and on the case wall. On the case wall enrichment information should be presented in the entity table form.
2. Enrichment table for the data that will be fetched from the target system should have keys with unique prefixes for grouping.
3. If there is a list in the response and action will use it for enrichment, then this list should be replaced as a comma-separated string. The only exception that can be done here is that for values that utilize ",", we can do the approach with multiple keys.
4. Action should return information in json result in Snake_Case.
5. If possible, action should have a logic for "is_suspicious" entity attribute, consider not add is_suspicious to host entities or analogs (because they shared across cases in Siemplify).
6. For each separate entity, action would need to create a separate insight.

2.3.5 Action Execution Flow

1. Action should return is_success=True if action run successfully and enriched at least one of the provided entities. IF it failed to enrich all provided entities - is_success should be False.

2.3.6 Case Wall

1. Action should have enrichment table on the case wall for user to see the enrichment made on entities.
2. Enrichment Case Wall table should be “entity table” type.
3. Case Wall table should not have preffixes added to enriched entity keys (prefixes should be in the json result).
4. If possible, a link that points to the same entity in the target system should be added to the entity table.
5. Enrichment table should have keys and related values in ROWS (not in columns), to “stretch“ the data vertically and not horizontally , to make the enrichment table more easier to read.
6. Enrichment table if possible should not have keys listed that dont have values to it, we should not show such keys.

2.3.7 Recommended Integrations to use for reference:

Windows Defender ATP - https://docs.google.com/document/d/1cixQO72Bz4qbXiCp-MGKCjLB31G4lHd5OqIaamqqi_g/edit#heading=h.pjbde6w1sd4c

2.4 "Execute Query" Actions Best Practices

2.4.1 Action Inputs

1. For each “Execute Query” action, there should also be the same action that utilizes the entities, "Execute Entity Query". Without it, using entities in Execute Queries actions can either impossible or very not user friendly.
2. Integration should have as a simple action that would allow user to construct a query - Execute Simple Query.
3. If action provides advanced functionality to create queries it should be called "Execute Custom Search" or “Execute Advanced Query“.
4. For all “Execute Query” actions, there should be
   a. parameter to limit results;
   b. parameter to order the results by specific returned key;
   c. parameter (CSV) to return only specific columns in result;
   d. parameter to specify a timeframe for which to return data.

2.4.2 Action Outputs

1. Case wall table should be created dynamically based on the query results.

2.4.3 Case Wall

1. In case of error or success, action should write in both log and case wall message the actual query that was executed by the action.

2.4.4 Recommended Integrations to use for reference:

Splunk - https://docs.google.com/document/d/1hrm90GgSS5_K6tq3MVv1hQazmwSdyinWiwJX_5eXGy8/edit#heading=h.w87cdiv8j61

Qradar Simple AQL Query - https://docs.google.com/document/d/1jWypoe8yB3qHhPh0-rwZRk_k_m1DhJMABKcQqxlr9X8/edit

2.5 "Upload/Download" Actions Best Practices

2.5.1 Action Inputs

1. Upload and download actions should allow user to work with multiple absolute file paths.
2. Action inputs should have a boolean parameter called “Overwrite”, which dictates whether to overwrite a file or not (if it exists in the target folder). If overwrite is disabled and file already exists, action should fail, even if only one of the files is like that.

2.5.2 Action Outputs

1. For download actions, in the JSON result there should always be a key referencing the absolute path of the file that was downloaded. As a parameter, user should provide an absolute path to the folder.

2.5.3 Recommended Integrations to use for reference:

VMware Lastline: https://docs.google.com/document/d/17n_u85SV3idCsHZnI-o7JKs6MSo6Ml8p57QNei505m0/edit

2.6 "Update" Actions Best Practices

2.6.1 Action Description

1. If action has parameters that needs to be specified together for the update - action should specify it in the description.

2.6.2 Action Inputs

1. Update actions should never have a default values that has an impact on the result. The default values should always be neutral, for example: “Select One” as a default one. If “Select One” is specified, then that parameter will be ignored during action execution. If none of the parameters are selected, then action should fail and state that one of the parameters should be specified.
2. For update actions correct error handling of input parameters is very important, it should be defined carefully.
3. Custom fields, labels - input parameter should accept key:value objects in the format of JSON, that might not be the most user friendly way, but is less prone to issues and unexpected errors, especially when multiple values are provided (See BMC Helix Remedyforce "Update Record" for reference).

2.6.3 Action Execution Flow

1. If none of the action input parameters are selected, then action should fail and state that one of the parameters should be specified.

2.6.4 Recommended Integrations to use for reference:

Microsoft Defender ATP - https://docs.google.com/document/d/1cixQO72Bz4qbXiCp-MGKCjLB31G4lHd5OqIaamqqi_g/edit#heading=h.eo2scywql650
BMC Helix Remedyforce - https://docs.google.com/document/d/1tx0A8BA00qSqYc0du75T1AdtecKa0JxEUxPE2AwE_0k/edit#heading=h.wuttpichgyne

2.7 "Mission-Critical" Actions Best Practices

2.7.1 Naming

1. As action is important, starting from naming action should clearly define what action does.

2.7.2 Description

1. Action description should describe on what entity types action is running on (User, Host, IP,…).
2. It is recommended to describe if the action performs action right away in target system or creates a task to make a change (Since because its a task, we cant be 100% sure action will ran on actual endpoint, it can be unreachable, turned off and so on).
3. If there are any notes, that User needs to be aware, for example something related to the action logic, action should have it in the description.

2.7.3 Action Inputs

1. Action should work on Entities in most cases, unless there is a specific use case to work based of action input parameters.

2.7.4 Action Execution Flow

1. Action should return is_success=True if action ran successfully on EVERY provided entity. If at least one of the provided entities failed to be processed by the action - **action should return is_success False**.

2.7.5 Recommended Integrations to use for reference:

Microsoft Defender ATP - https://docs.google.com/document/d/1cixQO72Bz4qbXiCp-MGKCjLB31G4lHd5OqIaamqqi_g/edit#heading=h.eo2scywql650

2.8 "Async" Actions Best Practices

2.8.1 Action Description

1. Action should state in the description that action is async, because we dont have other way to show user that action is async. Also action can describe that timeout can be increased in IDE for async action.

2.8.2 Action Inputs

1. Action should have a checkbox input parameter to wait the final results for async action, meaning that action will not just state that the task was created on the server, but actually will wait for the final result of task execution. Integration also should have a standalone additional action that can be used to check the statuses of created tasks. (see DUO integration example).

2.8.3 Action Outputs

1. For async actions outputs can change as action progresses, it should be specifically checked that action returns the latest status in json result as it runs or finishes.
2. If possible action should have in JSON result info on failed entities, so analyst can see for which entities action failed, and at what stage the async action failed (see CB live response integration, specifically implementation for async actions).

2.8.4 Action Execution Flow

1. Async actions due to the fact that they could run for long periods of time, for example days, need to have a mechanism to recover from unexpected network/connectivity issues. Async action should have a mechanism to do a 5 retries in case connectivity lost to the target system (see Exchange Extension Pack for example of implementation).
2. As action is running, action should track the timeout (python process timeout) for it, and if the timeout approaches, action should “catch“ the related error from Platform and should return error with a type of Timeout and write in the case wall: “Action stopped because its reached timeout. Please increase the timeout value for the action in Siemplify IDE“.

2.8.5 Case Wall

1. Action should have multiple variants of the messages it will put on the case wall, to display if action is running/failed/succeeded. Case Wall output message should be updated accordingly as action is running. (see CB Live Response example).

2.8.6 Example integrations for reference

Duo Auth API Integration: https://docs.google.com/document/d/11XAPSvvhoR64eAWVJX-hJyny3musWF-9zX4RlPi_weM/edit

CB Live Response Integration: https://docs.google.com/document/d/1O4fzHW3-5ULHUp_yxl4hMcOP6677bMY_uyL1yciGJIg/edit

2.9 "Send Mail" or "Send Message" Actions Best Practices

2.9.1 Action Inputs

1. For send mail actions the following minimum of action input parameters should be provided:
    a. Subject;
    b. Send to;
    c. CC;
    d. BCC;
    e. Attachments Paths;
    f. Mail content(email body).

2.9.2 Action Outputs

1. For send actions, action should be able to return message id for the sent message in the json result, so additional actions can use it later (wait for mail for example).

2.9.3 Recommended Integrations to use for reference:

Microsoft Exchange Integration - https://docs.google.com/document/d/1PAWFH55YqFat9s77bXUKg30TpdduooOjcu75gUdDS4E/edit

2.10 "Wait for the Update" Actions Best Practices

2.10.1 Action Description

1. Action should be async, and tell user that action is async in the description.

2.10.2 Action Inputs

1. Action should be able to take previous action results as input (for example message id of sent mail).
2. Wait for Update can be "simple" or "advanced". Simple works with one specific attribute, Advanced can work on multiple attributes.
3. If multiple attributes are monitored, action can have a drop down configuring how wait should work - wait for update of at least one attribute, or update of all attributes.

2.10.3 JSON Result

1. Action should return JSON result with the actual (latest) information on the object it was waiting for update.

2.10.4 Recommended Integrations to use for reference:

Microsoft Exchange Integration - https://docs.google.com/document/d/1PAWFH55YqFat9s77bXUKg30TpdduooOjcu75gUdDS4E/edit

2.11 Connector Best Practises

2.11.1 Connector Name

Connector names should state exactly what kind of information is going to be pulled into Siemplify in regards to the original system. Example: pulling incidents → Incidents Connector. Additionally, there should be information about integration. Example: RSA Netwitness Platform - Incidents Connector.

2.11.2 Connector Description

Connector description should provide the high level information about the connector and also all of the related notes that are important for the user to know. Also, description should contain the name of the “key” that is used for whitelist/blacklist purposes. Example: Pull incidents into Siemplify. Note: “eventType” key is used in the whitelist/dynamic list logic.

Depending if whitelist is mandatory for the connector to work or the there is a complex logic behind (example QRadar connectors or Splunk Query Connector), there should be an additional explanation describing the logic of the whitelist. For example, “In the whitelist you need to provide all of the queries that need to be executed by the connector. If no queries are provided, connector won’t do anything.”

2.11.3 Connector Parameters Description

Connector parameter description should provide enough information for the user to understand, what that parameter does. Connector parameter description in the connectors should start with “Specify”. Boolean parameters, need to start with “If enabled”. If there is a limited amount of supported values, connector parameter should state all of them. Example: “Possible Values: Informational, Low...”

2.11.4 Connector Structure

Connector structure consists of the following parameters:

1. Platform mandatory fields: Product Field Name, Event Field Name, Environment Field Name, Environment Regex Pattern, Script Timeout (Seconds)
2. API Root/Server Address
3. Credentials (API Key/Token, Username + Password, Client ID + Client Secret)
4. Custom Filters (Severity Filter, Type Filter and so on)
5. Max Hours/Days Backwards. Indicates at the starting point for the connector ingestion.
6. Max {object name} To Fetch. How many records to process per 1 connector iteration
7. Use whitelist as a blacklist.
8. Verify SSL
9. Proxy settings. Proxy Server Address, Proxy Username, Proxy Password.
10. Certificate File (On-prem integrations)

2.11.5 **Siemplify Alert Structure**

Name	**Is mandatory**	**What To Provide**
Ticket ID	True	Points at the id of the object in the original system). Example: if in Splunk event id is 123, then TicketId should also be 123
DisplayId	True	Used to identify Siemplify Alert. Should be manually generated uuid. Should be unique.
Name	True	Alert Name, which is shown in the UI of Siemplify.
Reason	False	Additional field for the analyst. Indicates the reason why this alert was triggered.
Description	False	Additional field for the analyst. Provides more context about the alert.
DeviceVendor	True	Vendor of the product that generated the alert. In most cases, it’s a hardcoded value. Example: Microsoft
DeviceProduct	True	Inherits value from “Product Field Name” parameter in the connector configuration. **Must have a fallback value.** In most cases, fallback value is the name of the product. Example: Azure Security Center.
Priority	True	Sets the alert Severity. Logic for this parameter should be provided in the connector flow.
Rule Generator	True	Mandatory field for the platform. Should point to the rule/policy in the source system that created the alert.

SourceGroupingIdentifier	False	Another identifier that platform can use to group alerts into 1 case.
StartTime/Endtime	True	Timestamps for the Siemplify Alert.
Extensions	False	Additional keys for the Siemplify Alert. Should contain information about the top level Alert. Needed, when in the original system there is an hierarchy.
Attachments	False	Additional attachment related to the Siemplify Alert.

2.11.6 **General Fields**

Parameter Display Name	**Type**	**Is mandatory**	**Default**	**Description**
Product Field Name	String	True	Product Name	Enter the source field name in order to retrieve the Product Field name.
Event Field Name	String	True	eventtype	Enter the source field name in order to retrieve the Event Field name.
Environment Field Name	String	False		Describes the name of the field where the environment name is stored. If the environment field isn't found, the environment is the default environment.

Environment Regex Pattern	String	False		A regex pattern to run on the value found in the "Environment Field Name" field. Default is .* to catch all and return the value unchanged. Used to allow the user to manipulate the environment field via regex logic. If the regex pattern is null or empty, or the environment value is null, the final environment result is the default environment.
Script Timeout (Seconds)	Int	True	600	Timeout limit for the python process running the current script.
Verify SSL	String	False	True	If enabled, verify the SSL certificate for the connection to the {Product Name} server is valid.
Use whitelist as a blacklist	bool	False	False	If enabled, whitelist will be used as a blacklist.
Alert Field Name	String	False	False	Name of the key that should be used for Alert Name. If nothing or invalid value is provided, the connector will use "{fall back value or key}" as fallback.
Proxy Server Address	String	False	False	The address of the proxy server to use.

Proxy Username	String	False	False	The proxy username to authenticate with.
Proxy Password	Password	False	False	The proxy password to authenticate with.

2.11.7 **Custom Filters**

2.11.7.1 **Severity Filter**

If Severity is presented in the incremental way, it should be implemeted in the following manner.

Parameter Display Name	**Type**	**Is mandatory**	**Description**
Lowest Severity To Fetch	String	False	The lowest severity that will be used to fetch Alert. If nothing is provided, the connector will ingest alerts with all severities. Possible values: Low, Medium, High

If Severity can be “Unknown“ or N/A then we need to provide a different structure:

Parameter Display Name	**Type**	**Is mandatory**	**Description**
Severity Filter	CSV	False	A comma-separated list of severities that will be used to fetch alerts. If nothing is provided, the connector will ingest alerts with all severities. Alerts with “N/A” severity in {product name} will have “Informational” severity in Siemplify. Possible values: N/A, Low, Medium, High.

If severity in the source product is not defined, we put “Informational” severity in Siemplify Alert for it.

Error Handling

If the user provides at least one invalid value. The connector should stop at show the following message with log level “Error”:

Invalid value provided in the "{parameter name}". Possible values: {list of possible values}

2.11.7.2 Max {object} To Fetch

This parameter defines how many Siemplify Alerts should be created in 1 iteration. The name of the parameter is directly correlated with the name of the connector. If connector works with incidents, then the name of the parameter will be "Max Incident To Fetch"

Parameter Display Name	Type	Default Value	Is mandatory	Description
Max {object} To Fetch	Int	50	False	How many {object} to process per one connector iteration.

Logic Nuances

From API perspective by default the connector should send max(Max {object} To Fetch, 100), which means that we query at least 100 objects, but we still only create as many objects as it's specified in "Max {object} To Fetch" parameter. Note: if connector is heavy then the default value can be lower than 100, in order to make sure that the connector is stable.

Error Handling

If the user provides at least one invalid value. The connector should stop at show the following message with log level "Error":

Invalid value provided in the "Max {object} To Fetch". Only positive integers are allowed.

2.11.7.3 Max Hours/Days Backwards

This parameter defines when the connector should start ingesting data. If nothing is provided, the connector should start from the current time.

Parameter Display Name	Type	Default Value	Is mandatory	Description
Max Hours/Days Backwards	Int	1	False	How many hours/ days backwards to fetch alerts. If nothing is provided, the connector will start ingesting from current time.

Logic Nuances

Sometimes there are situations, where there is limitation from API in terms of how much time backwards you can go, in that case the description of the parameter should provide how much backwards it can go.

If the product returns too much data, then execution needs to be segmented into chunks. For example, instead of querying data from 48 hours ago till now, connectors can fetch from 48 to 36 hours first, so that the volume of data will be more reasonable.

2.11.8 Siemplify Events Nuances

Every Siemplify Event should provide enough data to be used for mapping. If the Siemplify Event doesn't have entities then in 99% cases, it will not be useful for the analyst. Additionally, events should contain at least 1 field that will contain time value.

Understanding different scenarios

2 levels of hierarchy
Product has 2 levels of alert hierarchy. For example, 1 alert can contain multiple events that relate to that alert. In this situation, all of the related events will become Siemplify Events. Information about the Source Alert should be added as Siemplify Alert Extentions. Additionally, if you need Source Alert data for mapping purposes then you can add 1 Siemplify Event containing Source Alert information.

Fields of Event contain lists of entities of the same type

Sometimes in the response we can have the following structure:

```
{
    "urls": [
        {
            "url": "example.1"
        },
        {
            "url": "example.2"
        },
        {
            "url": "example.3"
        },
        {
            "url": "example.4"
        }
    ]
}
```

In this situation, the connector would need to split one event from the source system into 4 Siemplify Events, so that the mapping can be done.

The Structure of JSON makes it hard to map entities

Sometimes, there can be a situation like this:

```
{
    "entities": [
        {
            "type": "url",
            "value": "example.1"
        },
        {
            "type": "hash",
            "value": "123123123123123"
        }
    ]
}
```

Platform won't be able to distinguish between different entity types, so it should be done by the connector. The structure should be transformed into the following thing:

```
{
    "entities": [
        {
            "type": "url",
            "value": "example.1",
            "url": "example.1"
        },
        {
            "type": "hash",
            "value": "123123123123123",
            "hash": "123123123123123"
        }
    ]
}
```

There are more than 200 events:

If the product can contain more than one 200 events, connector would need to create multiple Siemplify Alerts. Note: if there is a 2 layer hierarchy 1 Siemplify Event should contain information about 1 layer in all created Siemplify Alerts. This is needed to allow analyst to correlate multiple Siemplify Alerts to alert in the original system. Display ID should be unique between those splitted alerts and SourceGroupIdentifier should point to the ID of the original ticket. It will allow the platform to group those Siemplify Alerts under one case.

Siemplify event doesn't have a unique identifier:

If there is a situation, where response doesn't have a key that is unique, then the connector needs to take the SHA256 hash out of the whole JSON object and store it as id in the ids.json.

2.11.9 Handling Updates To Data

Sometimes in the source product, there can be a situation where the alert/event is updated in time. Unfortunately, Siemplify doesn't allow users to update Siemplify Events.

The best way to track updates is to check the amount of Siemplify Events related to the alert. In the *ids.json*, we will store the amount of events and then compare, if some updates were identified.

This is the structure of *ids.json*:

“{id}:{count events}” -> “123:10”

In order to consistently track updates, API should return data based on “lastUpdateTime” instead of “createdTime”.

If an update is identified, action should create a new Siemplify Alert with all of the necessary data.

2.11.10 **Product Field Name and Event Field Name nuances**

Make sure that the default value provided for these two parameters point to a value. This is needed, so that in mapping screen, users won't see "Unknown event/product name".

For product field name, if there is no good field just put "Product Name". Additionally, make sure that there is a fallback value, so that even if the user provides something that is not valid, it would still have some sort of value. By default the fallback value is the same as "Product Name".

2.11.11 **General Error Handling**

If parameter works with 1 predefined value:

This is a situation can be seen with "Lowest Severity To Fetch" parameters. If we know beforehand all of the possible values, then if user provides an invalid one, connector should immediately stop and raise an error. Structure of the error should look like this:

Invalid value provided in the "{parameter name}". Possible values: {list of possible values}

If parameter works with multiple predefined values:

This situation can be seen frequently with some custom filters, for example, "Type Filters" or "Status Filters". If we know beforehand all of the possible values then there are two cases that need to be handled.

1 invalid value is provided:

Invalid value provided in the "{parameter name}". Possible values: {list of possible values}

at least one correct value is provided:

In this situation, we don't stop the connector just leave a "Warning" log stating that connector is ignoring the invalid values and will only handle the correct ones.

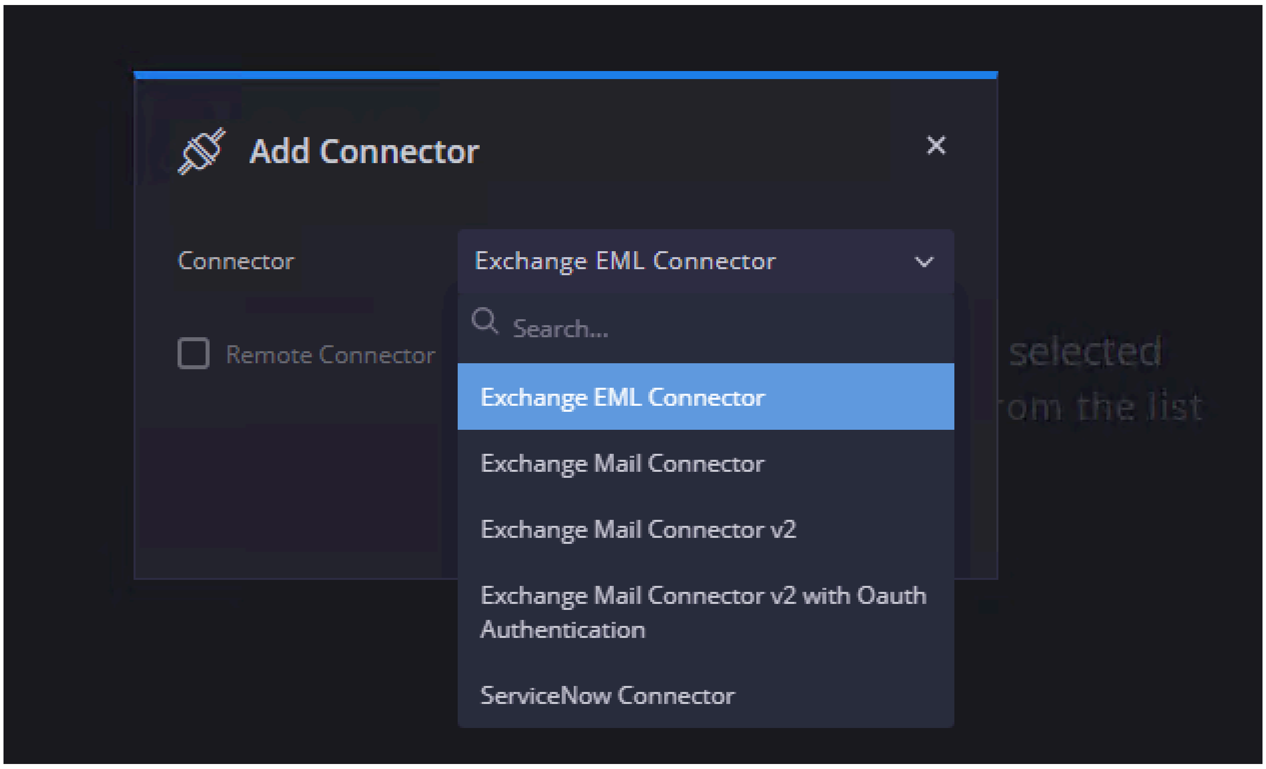
Example of the log message can be the following:

The following values for the parameter "parameter name" are invalid: {csv of invalid values}. Ignoring them. Possible values: {list of possible values}

2.12 Connector Best Practices v1

2.12.1 **Connector Name**

1. Connector names should state exactly what kind of information is going to be pulled into Siemplify in regards to the original system, additionally, integration name should be added to connector name as well, for User to easily see to which integration connector belongs to:



Connector name example :

```
RSA Netwitness Platform - Incidents Connector.
```

2.12.2 **Connector Description**

1. Connector description should provide the high level information about the connector and also all of the related notes that are important for the user to know. Also, description should contain the name of the “key” that is used for whitelist/blacklist purposes. Example:

   ```
   Pull incidents into Siemplify.
   Note: “eventType” key is used in the whitelist/dynamic list logic.
   ```

2. Depending if the whitelist is mandatory or not for the connector to work, or if there is a complex logic behind (example QRadar connectors or Splunk Query Connector), there should be an additional explanation describing the logic of the whitelist. For example:

   ```
   “In the whitelist you need to provide all of the queries that need to be
   executed by the connector.
   If no queries are provided, connector won’t do anything.”
   ```

2.12.3 Connector Parameters Description

1. Connector parameter description should provide enough information for the user to understand, what that parameter does.
2. Connector parameter description in the connectors should start with “Specify”.
3. Boolean parameters description should start with “If enabled”.
4. If there is a limited amount of supported values, connector parameter should state all of them. Example:

```
“Possible Values: Informational, Low...”
```

2.12.4 Common Connector Parameters

1. Connector usually have the following parameters available for the User to configure, in the next sections specific parameters will be discussed in detail:
   a. Platform mandatory fields: Product Field Name, Event Field Name, Environment Field Name, Environment Regex Pattern, Script Timeout (Seconds)
   b. API Root/Server Address
   c. Credentials (API Key/Token, Username + Password, Client ID + Client Secret)
   d. Custom Filters (Severity Filter, Type Filter and so on)
   e. Max Hours/Days Backwards. Indicates at the starting point for the connector ingestion.
   f. Max {object name} To Fetch. How many records to process per 1 connector iteration
   g. Use whitelist as a blacklist.
   h. Verify SSL
   i. Proxy settings. Proxy Server Address, Proxy Username, Proxy Password.
   j. Certificate File (On-prem integrations)

Example of the connector configuration parameters as they are defined in specification:

Parameter Display Name	Type	Is mandatory	Default	Description
Product Field Name	String	True	Product Name	Enter the source field name in order to retrieve the Product Field name.
Event Field Name	String	True	eventtype	Enter the source field name in order to retrieve the Event Field name.

Environment Field Name	String	False		Describes the name of the field where the environment name is stored. If the environment field isn't found, the environment is the default environment.
Environment Regex Pattern	String	False		A regex pattern to run on the value found in the "Environment Field Name" field. Default is .* to catch all and return the value unchanged. Used to allow the user to manipulate the environment field via regex logic. If the regex pattern is null or empty, or the environment value is null, the final environment result is the default environment.
Script Timeout (Seconds)	Int	True	600	Timeout limit for the python process running the current script.
Verify SSL	String	False	True	If enabled, verify the SSL certificate for the connection to the {Product Name} server is valid.
Use whitelist as a blacklist	bool	False	False	If enabled, whitelist will be used as a blacklist.

Alert Field Name	String	False	False	Name of the key that should be used for Alert Name. If nothing or invalid value is provided, the connector will use "{fall back value or key}" as fallback.
Proxy Server Address	String	False	False	The address of the proxy server to use.
Proxy Username	String	False	False	The proxy username to authenticate with.
Proxy Password	Password	False	False	The proxy password to authenticate with.

2.12.5 Connector Product Field Name and Event Field Name parameters nuances

1. Since both Product Field Name and Event Field Name are used in the mapping process of the original system attributes to Siemplify attributes, the default value provided for these two parameters should point to a key that have a corresponding value. This is needed to avoid situations when Users see "Unknown event/ product name" on the event mapping screen in Siemplify.
2. For product field name, if there is no good field to take from event data, value for the key used for Product Field Name can have just a product name, like "FireEye HX" or "Splunk ES".
3. For product field name, make sure that there is a fallback value, so that even if the user provides something that is not valid, it would still have some sort of value. By default the fallback value is the same as Product Name, like "FireEye HX" or "Splunk ES".

2.12.6 Connector parameters for filtering.

1. Connector configuration parameters usually include "custom" filters - parameters, that allow User to configure the attributes, based on which alert from the original system will be loaded in Siemplify or not.

2.12.6.1 Severity Filter.

1. Severity filter is one of the examples of the "custom" filters that Users can configure to filter out alerts from the original system.
2. If in original system severity is presented in the incremental way, it should be implemeted in the following manner:

Parameter Display Name	**Type**	**Is mandatory**	**Description**

Lowest Severity To Fetch	String	False	Specify the lowest severity that will be used to fetch Alert. If nothing is provided, the connector will ingest alerts with all severities. Possible values: Low, Medium, High

3. If severity can be "Unknown" or N/A then we need to provide a different structure:

Parameter Display Name	**Type**	**Is mandatory**	**Description**
Severity Filter	CSV	False	Specify a comma-separated list of severities that will be used to fetch alerts. If nothing is provided, the connector will ingest alerts with all severities. Alerts with "N/A" severity in {product name} will have "Informational" severity in Siemplify. Possible values: N/A, Low, Medium, High.

4. If severity in the source product is not defined, we put "Informational" severity in Siemplify Alert for it.

5. Error Handling for the Severity Filter. If the user provides at least one invalid value. The connector should stop at show the following message with log level "Error":

```
Invalid value provided in the "{parameter name}".
Possible values: {list of possible values}
```

2.12.7 **Max {objects} To Fetch.**

1. Max {objects} To Fetch parameter defines how many Siemplify Alerts should be created in 1 iteration. The name of the parameter is directly correlated with the name of the connector. If connector works with incidents, then the name of the parameter will be "Max Incident To Fetch".

Example "Max {objects} To Fetch" parameter snippet for specification:

Parameter Display Name	**Type**	**Default Value**	**Is mandatory**	**Description**

Max {objects} To Fetch	Int	50	False	Specify how many {object} to process per one connector iteration.

2. Max {objects} To Fetch Logic Nuances. From API perspective by default the connector should send max(Max {object} To Fetch, 100), which means that we query at least 100 objects, but we still only create as many objects as it's specified in "Max {object} To Fetch" parameter. Note: if connector is heavy then the default value can be lower than 100, in order to make sure that the connector is stable.

3. Max {objects} To Fetch Error Handling. If the user provides at least one invalid value, the connector should stop its execution and print the following message in the connector log with log level "Error":

```
Invalid value provided in the "Max {object} To Fetch".
Only positive integers are allowed.
```

2.12.7.1 **Max Hours/Days Backwards.**

1. Max Hours/Days Backwards parameter defines when the connector should start ingesting data. If nothing is provided, the connector should start from the current time.

Example "Max Hours/Days Backwards" parameter snippet for specification:

Parameter Display Name	**Type**	**Default Value**	**Is mandatory**	**Description**
Max Hours/Days Backwards	Int	1	False	Specify how many hours/days backwards to fetch alerts. If nothing is provided, the connector will start ingesting from current time.

2. Max Hours/Days Backwards Logic Nuances. If in the original system there is an API limitation for the maximum time backwards API can return data for, Max Hours/Days Backwards parameter description should include information for how much backwards connector can go.

3. If the original system can return substantial for processing ammount of data with default Max Hours/Days Backwards values, then the connector execution should to be segmented into chunks - instead of querying data from 48 hours ago till current time, connector should fetch data from 48 to 36 hours first, when from 36 to 24, and so on.

2.12.8 **General Error Handling in the Connector Parameters**

1. If the connector parameter works with a single predefined value and all of the possible values are known, then if user provides an invalid one, connector should immediately stop its execution and raise an error. This is a situation that can be seen with "Lowest Severity To Fetch" parameters. Structure of the error to add to the connector log should look like this:

```
Invalid value provided in the “{parameter name}”.
Possible values: {list of possible values}
```

2. If the connector parameter works with multiple predefined values and all of the possible values for the parameter are known, then there are two cases that need to be handled with different error messages, but connector should stop its execution in both cases. This situation can be seen frequently with some custom filters, for example, “Type Filters” or “Status Filters“.

If a single invalid value is provided, the error to add to the connector log should look like this:

```
Invalid value provided in the “{parameter name}”.
Possible values: {list of possible values}
```

If at least one correct value is provided, the connector execution shouldn’t be stopped, the connector should continue to work with printing to the log a message that that connector is ignoring the invalid values and will only handle the correct ones. Example of the log message can be the following:

```
The following values for the parameter “parameter name“ are invalid: {csv of invalid
values}.
Ignoring them.
Possible values: {list of possible values}
```

2.12.9 **Siemplify Alert**

1. Siemplify alert created by the connector should has a strict structure that connectors should follow. The following table contain all possible alert attributes with information on them:

Name	**Is mandatory**	**What To Provide**
Ticket ID	True	Points at the id of the object in the original system). Example: if in Splunk event id is 123, then TicketId should also be 123
DisplayId	True	Used to identify Siemplify Alert. Should be manually generated uuid. **Should be unique for every created alert.**
Name	True	Alert Name, which is shown in the UI of Siemplify.
Reason	False	Additional field for the analyst. Indicates the reason why this alert was triggered.

Description	False	Additional field for the analyst. Provides more context about the alert.
DeviceVendor	True	Vendor of the product that generated the alert. In most cases, it's a hardcoded value. Vendor field is used in Siemplify Ontology to define the event mapping hierarchy. Example: Microsoft
DeviceProduct	True	Inherits value from "Product Field Name" parameter in the connector configuration. **Must have a fallback value.** In most cases, fallback value is the name of the product. Example: Azure Security Center.
Priority	True	Sets the alert Severity. Logic for this parameter should be provided in the connector flow.
Rule Generator	True	Mandatory field for the platform. Should point to the rule/policy in the source system that created the alert.
SourceGroupingIdentifier	False	Another identifier that platform can use to group alerts into 1 case.
StartTime/Endtime	True	Timestamps for the Siemplify Alert.
Extensions	False	Additional keys for the Siemplify Alert. Should contain information about the top level Alert. Needed, when in the original system there is an hierarchy.

Attachments	False	Additional attachment related to the Siemplify Alert. Can be used to add to the Siemplify Case Wall original object that caused the alert. For example, it is used in Exchange/Email integrations to attach to the Case Wall received email file EML for the ingested alert.

2.12.10 **Siemplify Event**

1. Every Siemplify Event should provide enough data to be used for mapping of data from original system to siemplify. If the Siemplify Event doesn't have entities then in 99% cases, it will not be useful for the analyst.
2. Every Siemplify Event should contain at least one field that will contain time value.
3. Additional logic for Siemplify event should be added if the product has 2 levels of alert hierarchy - for example, one alert can contain multiple events that relate to that alert. In this situation, all of the related events will become Siemplify Events. Information about the Source Alert should be added as Siemplify Alert Extentions. Additionally, if you need Source Alert data for mapping purposes then you can add one Siemplify Event containing Source Alert information.
4. Fields of Event contain lists of entities of the same type. Sometimes in the response we can have the following structure:

```
{
    "urls": [
        {
            "url": "example.1"
        },
        {
            "url": "example.2"
        },
        {
            "url": "example.3"
        },
        {
            "url": "example.4"
        }
    ]
}
```

In this situation, the connector would need to split one event from the source system into four Siemplify Events, so that the mapping can be done.

5. The Structure of JSON makes it hard to map entities. If the events in the original system have the following structure:

```
{
    "entities": [
        {
            "type": "url",
            "value": "example.1"
        },
        {
            "type": "hash",
            "value": "123123123123123"
        }
    ]
}
```

Platform won't be able to distinguish between different entity types, so it should be done by the connector. The structure should be transformed into the following thing:

```
{
    "entities": [
        {
            "type": "url",
            "value": "example.1",
            "url": "example.1"
        },
        {
            "type": "hash",
            "value": "123123123123123",
            "hash": "123123123123123"
        }
    ]
}
```

6. If the product can contain more than 200 events, and there is a need to ingest all of the events, the connector should create multiple Siemplify Alerts . Note: if there is a 2 layer hierarchy 1 Siemplify Event should contain information about 1 layer in all created Siemplify Alerts. This is needed to allow analyst to correlate multiple Siemplify Alerts to alert in the original system. Display ID should be unique between those splitted alerts and SourceGroupIdentifier should point to the ID of the original ticket. It will allow the platform to group those Siemplify Alerts under one case.

7. If an event from the original system doesnt have a field that the connector can use for the unique identifier, then the connector needs to take the SHA256 hash out of the whole JSON object and store it as the unique identifier (id) in the ids.json.

2.12.11 Handling Updates of alert or event data in the original system.

1. If alert or event data can be updated over the time after they are ingested to Siemplify additional logic should be implemented in the connector as Siemplify doesn't allow users to update Siemplify Alerts or Events.
2. The best way to track such updates is to check the amount of Siemplify Events related to the alert. In the *ids.json*, we will store the amount of events and then compare, if some updates were identified. Example structure of *ids.json*:

```
“{id}:{count events}” -> “123:10”
```

3. In order to consistently track updates, API should return data based on “lastUpdateTime” instead of “createdTime”.
4. If an update is identified, action should create a new Siemplify Alert with all of the necessary data.