

WHITE PAPER: January 2025



Beginners Guide to Automation

with vDefend Firewall

Table of contents

Overview	3
What are CRUD actions?.....	4
Ways to Automate.....	4
REST Calls using the NSX Policy API	4
Terraform and OpenTofu	4
Ansible	5
PowerCLI	5
Aria Automation	5
SDK	6
Policy-Based API Fundamentals.....	6
Hierarchical API Structure	6
Hierarchical API Call	6
Singular Calls	8
Cursor/Paging	9
Sequence Numbers	10
Authentication	14
Rate Limiting	17
Full Scale Automation Example	18
Gather VM Information:	18
Groups	21
Custom Services	26
Policy	28
Hierarchical API Example:	34
Summary	39
Helpful Links and Coding Examples:	40

Revision History

Author	Version	Last Updated
Andrew Hrycaj	1.0	1/28/2025

Overview

vDefend security automation with NSX can be achieved using many different languages and tools. Today, automation is required by operations teams to manage large deployments of IT assets and their configurations. From Infrastructure as Code (IaC) to Self-Service, these automation strategies are used to provide both operations teams and end users the ability to create, read, update, and delete (CRUD) these assets. In this paper, we discuss the different languages and tools available for NSX automation along with key topics of interest to consider when using the NSX Policy API along with detailed examples.

What are CRUD actions?

Create, read, update, and delete (CRUD) actions are executed using REST methods. There are several REST methods you can use to perform changes using an endpoint's API. While this document is not intended to serve as a definitive guide to REST, a quick definition of the methods are defined below:

- **DELETE:** Removes a resource from the server database
- **GET:** Retrieves information about a resource without modifying it
- **POST:** Creates a new resource in a collection, associating it with the parent resource
- **PUT:** Updates an existing resource on the server
- **PATCH:** Sends a request body that specifies which fields to update

You should always refer to the API documentation for a full understanding of what these methods will perform when coupled with an API call.

Ways to Automate

REST Calls using the NSX Policy API

You can utilize the NSX Manager Policy API to execute CRUD actions against the NSX Manager. This method allows you to write your own customized scripts that meet your exact automation needs without any potential coding overhead. Languages like Python, GoLang, Javascript and more all have libraries that allow you to make HTTP/HTTPS REST calls to interact with the NSX Manager Policy API. You can create single execution scripts to perform a task once or create more complex scripts which are idempotent that allow you to run them multiple times and ensure you are not duplicating creation or deletions of objects.

Creating your own self-service portals using a scripting language and NSX Manager Policy API is also a common use-case. Custom web frontends, data processing and governance backends, logging, CMDB interaction, and more can be designed and deployed as microservices for redundancy and scalability. These options are for more advanced use-cases and often require a dedicated and experienced infrastructure software development team.

Terraform and OpenTofu

Terraform and Opentofu are used for infrastructure as code (IaC) automation strategies. IT operations teams, product owners and developers will use IaC to deploy their workloads and configure the infrastructure to meet the needs of the application. From a network and security standpoint, this would include load-balancer, firewall, and advanced threat prevention (ATP) configurations. A manifest is created with all the required workloads and configuration variables. When the deployment is initially executed, all the workloads and components will be configured and their state stored in a state file. Any changes to the manifest will result in workload or configuration changes in the infrastructure to reflect what is in the manifest file. Any deletions from the file will also remove the workload or configuration from the infrastructure.

Infrastructure as code can be coupled with continuous integration/continuous deployment pipelines (CI/CD) in a repository like Github or Gitlab. The manifest file containing all of the workload and infrastructure configuration can be kept track of with normal git methods. Changes can then be promoted to the main branch and terraform/opentofu will perform the changes via the pipeline. This strategy mimics the software development process that developers are already comfortable with.

Ansible

Ansible is mostly used to provision the NSX core components like NSX Managers, Edges, and Transport Nodes while configuration is handled by Self-Service portals or Terraform/Opentofu. Ansible can also be used to configure the infrastructure to meet the application's network and security needs. Both strategies are commonly used by customers so it is your choice which you would like to use. Like any other automation planning, use the tools that best fit the needs of your IT operations and business teams.

PowerCLI

PowerCLI is an additional way to automate CRUD actions against an NSX Manager. The module is located [here](#). This is a popular option because of its simplistic scripting syntax and widespread use. There are also numerous modules for other VMware products which makes it ideal when creating complex automation scripts that must reach multiple types of endpoints. PowerCLI can be used on both Linux and Windows systems.

Aria Automation

Aria Automation includes a suite of tools that allow you to create complex automation tasks using a built in UI and code. There are many components which are described below:

Aria Assembler

You use VMware Aria Automation Assembler to connect to your public and private cloud providers so that you can deploy machines, applications, and services that you create to those resources. You and your teams develop cloud-templates-as-code in an environment that supports an iterative workflow, from development to testing to production. At provisioning time, you can deploy across a range of cloud vendors.

Aria Orchestrator

VMware Aria Automation Orchestrator is a workflow automation solution designed to simplify the automation of complex IT tasks. It helps improve service delivery efficiency, operational management, and IT agility. VMware Aria Automation Orchestrator is built with an open and flexible architecture that system administrators and IT operations staff can use to streamline tasks and integrate functions with third-party software through workflows.

Aria Service Broker

The VMware Aria Automation Service Broker provides a single point where you can request and manage catalog items. As a cloud administrator, you create catalog items by importing released VMware Aria Automation Assembler cloud templates and Amazon Web Services CloudFormation templates that your users can deploy to your cloud vendor regions or datastores.

As a user, you can request and monitor the provisioning process. After deployment, you manage the deployed catalog items throughout the deployment lifecycle.

Aria Automation Pipeline

Automation Pipelines models the tasks in your software release process, automates the development and test of developer code, and releases it to your production environment. It integrates your release process with developer tools to accomplish specific tasks, and tracks all code artifacts and versions.

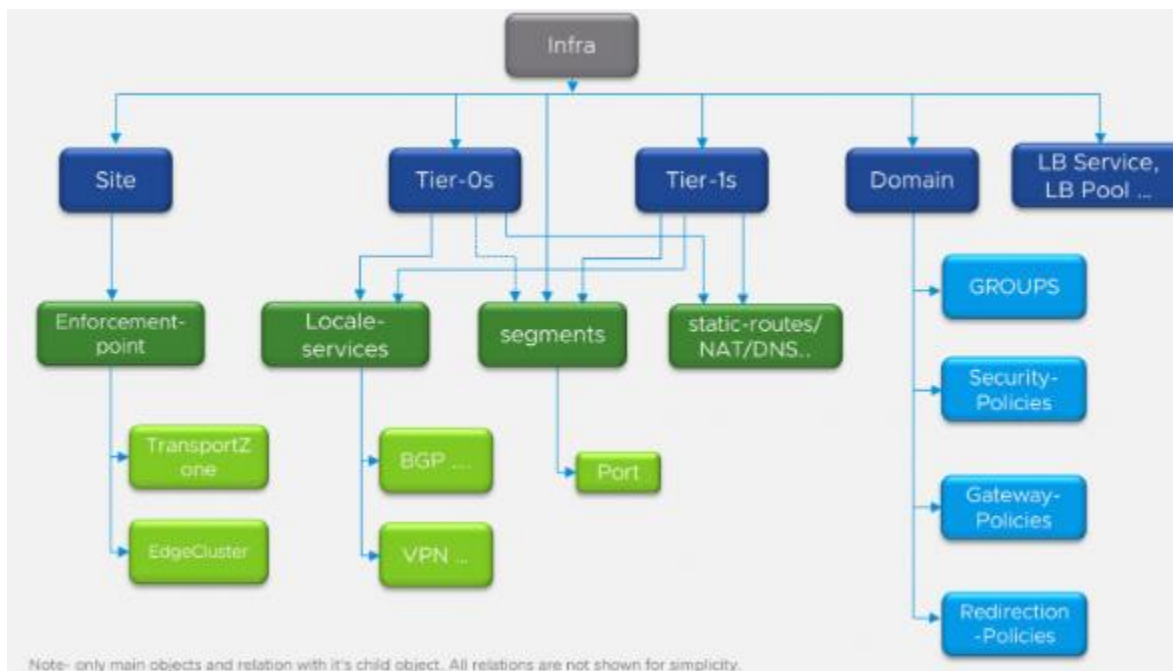
All of the Aria Automation components provide a framework that helps individuals automate their on-prem enterprise private cloud. These components come with your VMware Virtual Cloud Foundation (VCF) license along with many other helpful tools included in the VMware Aria Operations Suite.

SDK

The VMware SDK for Java, Python, and GoLang are mostly used for development of Terraform and Ansible modules. While this is an option, it is not one that is utilized by many. The latest version of the Python SDK is 4.0.1 as of this writing and is located [here](#). The Java SDK's latest version is also 4.0.1 as of this writing and is located [here](#).

Policy-Based API Fundamentals

Hierarchical API Structure



The hierarchical tree structure is made of nodes and have parent/child, peer relationships with each other. The API follows the tree structure above when performing CRUD actions. All policy API calls will start with `/policy/api/v1` followed by the tree flow. For example, when wanting to make CRUD for groups, you would reference `/infra/domain/<domain_name>/groups`. Calls to objects that are not associated with projects will be created under the default domain. So, the full URI for our call would be:

```
https://<NSX_MANAGER_FQDN_OR_IP>/policy/api/v1/infra/domain/default/groups
```

If we follow the tree for service policies, the path would be `/infra/domain/default/security-policies`. As long as you understand the hierarchical tree flow, you can easily understand and read the API documentation.

Hierarchical API Call

The hierarchical API allows you to define all aspects of a network and security configuration in a single REST call. NSX Segment, Groups, Services, Policy, Rules and more can all be defined and easily understood since all information is in the payload. The policy is defined differently from a singular call. The URI for this type of call is listed below:


```
/policy/api/v1/infra
```

From here, we can use a singular payload to define many different objects within the NSX Manager. In the example below, we create two groups, a security policy and a singular rule utilizing the two new groups we create in the same payload.

```
{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildDomain",
      "marked_for_delete": false,
      "Domain": {
        "id": "default",
        "resource_type": "Domain",
        "marked_for_delete": false,
        "children": [
          {
            "resource_type": "ChildGroup",
            "Group": {
              "id": "WebGroup",
              "resource_type": "Group",
              "display_name": "WebGroup"
            }
          },
          {
            "resource_type": "ChildGroup",
            "Group": {
              "id": "AppGroup",
              "resource_type": "Group",
              "display_name": "AppGroup"
            }
          }
        ]
      }
    },
    {
      "resource_type": "ChildSecurityPolicy",
      "marked_for_delete": false,
      "SecurityPolicy": {
        "id": "HierarchicalPolicy",
        "resource_type": "SecurityPolicy",

```

```
"display_name": "Hierarchical Policy",
"description": "Policy created by Hierarchical API",
"rules": [
  {
    "resource_type": "Rule",
    "id": "Rule-1",
    "description": "Rule created by Hierarchical API",
    "display_name": "Web to App rule",
    "sequence_number": 10,
    "source_groups": [
      "/infra/domains/default/groups/WebGroup"
    ],
    "destination_groups": [
      "/infra/domains/default/groups/AppGroup"
    ],
    "services": [
      "/infra/services/HTTPS"
    ],
    "action": "ALLOW",
    "scope": [
      "/infra/domains/default/groups/WebGroup",
      "/infra/domains/default/groups/AppGroup"
    ]
  }
]
}
}
}
}
}
}
}
}
}
}
```

Singular Calls

Another way to create objects in the NSX Manager is to make a simple singular API call with a payload specifically for the object you wish to create. This can be a security group, service, policy, rule and more. The URI for this type of call will be

directed to the exact object you wish to create within the Hierarchical API structure. Singular calls are best used for one-time CRUD actions.

In this example, we will create a security group called "DB_Servers". The URI for this call is listed below:

```
/policy/api/v1/infra/domains/default/groups/{ID}
```

The ID for the call will be "DB_Server". This means when the object is created, it will use the ID of DB_Server. We can use that in subsequent API calls to change (PATCH) information, read or delete the group. The payload will be much shorter and simpler since we are only creating one group:

```
{
  "description": "DB_Servers",
  "display_name": "DB_Servers",
  "expression": [
    {
      "member_type": "VirtualMachine",
      "value": "DB_Servers",
      "key": "Tag",
      "operator": "EQUALS",
      "resource_type": "Condition"
    }
  ]
}
```

The expression is used to match VMs with a tag of DB_Servers so they will automatically be added to the group. More detailed information about how expressions work can be found in the [API documentation](#).

Cursor/Paging

The NSX Policy API allows you to perform CRUD actions for any object in the manager. When retrieving information using a GET call, there might be a large amount of data that needs to be returned. Large amounts of data can take time to gather in the NSX Manager and can cause long delays in response. In order to avoid this type of situation, NSX uses cursor pagination to help break up the response payload into smaller chunks for quicker response times. You can think of a cursor value as a bookmark of where data retrieval will start when making a GET call.

By default, NSX will return 1000 entries before it will use a cursor. This means if your call has less than 1000 entries, the payload with the data will not include a cursor value since you have all the data. We will now look at an example of grabbing more than 1000 vDefend groups from the NSX Manager.

```
{
  "results": [
    {
      <--Omitted-->
    }
  ],
  "result_count": 5024,
```

```
"sort_by": "create_time",
"sort_ascending": true,
"cursor": "0076/infra/domains/default/groups/autogroup-cd410cec-2bdc-44ac-b44a-
b0aa0064c38eMTczNTY2NDI1NTE5OQ=="
}
```

The cursor value above indicates we have more than a 1000 groups in this NSX Manager, so it only returned the first 1000 groups. If we want to get the next 1000 groups, we would need to include the cursor value as a parameter in the GET call so the NSX Manager will start gathering information from that starting point. An example of that call is below:

```
https://<nsxmanager_IP_OR_FQDN>/policy/api/v1/infra/domains/default/groups?cursor=0076/infra/domains/default/gr
oups/autogroup-cd410cec-2bdc-44ac-b44a-b0aa0064c38eMTczNTY2NDI1NTE5OQ==
```

Once this call is made, it will return the next round of results:

```
{
  "results": [
    {
<--Omitted-->
    },
    {
      "sort_by": "create_time",
      "sort_ascending": true,
      "cursor": "0076/infra/domains/default/groups/autogroup-bb1456ff-c96f-4c90-b8aa-
e6563f130a66MTczNTY2NDMyNjg2OAA=="
    }
  ]
}
```

A different cursor value has been included in this payload indicating we have even more vDefend groups in the NSX Manager. We would continue in a loop with a programming language until we retrieve all of the vDefend groups in the NSX Manager. When the payload does not have a cursor value in the return, we know there are no more groups to retrieve from the NSX Manager.

Sequence Numbers

Like all firewalls, rules in the Distributed Firewall are evaluated from top-down. The highest priority or lowest sequence number is evaluated first. If a match is found, processing stops and the action of the rule is executed. If there is no match, then processing continues until the default rule is reached. Here is an example of two rules inside a policy called "App_Policy". Lots of information has been omitted in the calls so we can focus on the sequence numbers

```
{
  "rules": [
    {
      "id": "Web_rule",
      "rule_id": 10216,
      "sequence_number": 10,
    },
  ]
}
```

```
{
  "id": "Web_to_DB",
  "rule_id": 10217,
  "sequence_number": 20,
}
],
"resource_type": "SecurityPolicy",
"sequence_number": 2,
"internal_sequence_number": 60000002,
}
```

Web_rule has a sequence number of 10 which, in this case, will make it the first rule to be evaluated within the 'App_Policy' policy. If we wanted a rule to be inserted between the two, we would create a new firewall rule with a sequence number of 15. This can be achieved in the API by specifying the 'sequence_number' in the payload of the new rule or by dragging the new rule between the two rules using the GUI.

Example:

```
{
  "action": "ALLOW",
  "resource_type": "Rule",
  "id": "New_rule",
  "display_name": "New rule",
  "sequence_number": 15,
  "source_groups": [
    "/infra/domains/default/groups/All_Web"
  ],
  "destination_groups": [
    "/infra/domains/default/groups/App_Servers"
  ],
  "services": [
    "/infra/services/SSH"
  ],
  "profiles": [
    "ANY"
  ],
  "logged": true,
  "scope": [
    "/infra/domains/default/groups/All_Web",
    "/infra/domains/default/groups/App_Servers"
  ],
  "disabled": false,
}
```

```
    "notes": ""  
  }  
}
```

If we create a new rule without specifying a sequence number, the NSX Manager will assign it a value 0. If we have multiple rules with the same sequence number, the order will not be deterministic. Please refer to the [API documentation](#) for more information. Part of the API documentation is listed below:

If there are multiple rules with the same sequence number then their order is not deterministic. If a specific order of rules is desired, then one has to specify unique sequence numbers or use the POST request on the rule entity with a query parameter `action=revise` to let the framework assign a sequence number

Example with no sequence number:

```
{  
  "action": "ALLOW",  
  "resource_type": "Rule",  
  "display_name": "New_rule",  
  "source_groups": [  
    "/infra/domains/default/groups/VDI"  
  ],  
  "destination_groups": [  
    "/infra/domains/default/groups/WebServer"  
  ],  
  "services": [  
    "/infra/services/SSH"  
  ],  
  "profiles": [  
    "ANY"  
  ],  
  "logged": true,  
  "scope": [  
    "/infra/domains/default/groups/WebServer",  
    "/infra/domains/default/groups/VDI"  
  ],  
  "disabled": false  
}
```

Result:

```
{  
  "action": "ALLOW",  
  "resource_type": "Rule",
```

```
"id": "new_rule",
"display_name": "New_rule",
"path": "/infra/domains/default/security-policies/App_Policy/rules/new_rule",
"relative_path": "new_rule",
"parent_path": "/infra/domains/default/security-policies/App_Policy",
"unique_id": "00000000-0000-0000-0000-000000010218",
"realization_id": "00000000-0000-0000-0000-000000010218",
"owner_id": "46d1e70-1a94-415e-93de-70011f9f6b28",
"marked_for_delete": true,
"overridden": false,
"rule_id": 10218,
"sequence_number": 0,
"sources_excluded": false,
"destinations_excluded": false,
"source_groups": [
  "/infra/domains/default/groups/VDI"
],
"destination_groups": [
  "/infra/domains/default/groups/WebServer"
],
"services": [
  "/infra/services/SSH"
],
"profiles": [
  "ANY"
],
"logged": true,
"scope": [
  "/infra/domains/default/groups/WebServer",
  "/infra/domains/default/groups/VDI"
],
"disabled": false,
"direction": "IN_OUT",
"ip_protocol": "IPV4_IPV6",
"is_default": false,
"_system_owned": false,
"_protection": "NOT_PROTECTED",
"_create_time": 1735670391083,
"_create_user": "admin",
"_last_modified_time": 1735670391083,
```

```
"_last_modified_user": "admin",
"_revision": 0
}
```

Policies have both 'sequence_number' and 'internal_sequence_number' for ordering. Please refer to the [API documentation](#) for more information about how they work with categories and domains.

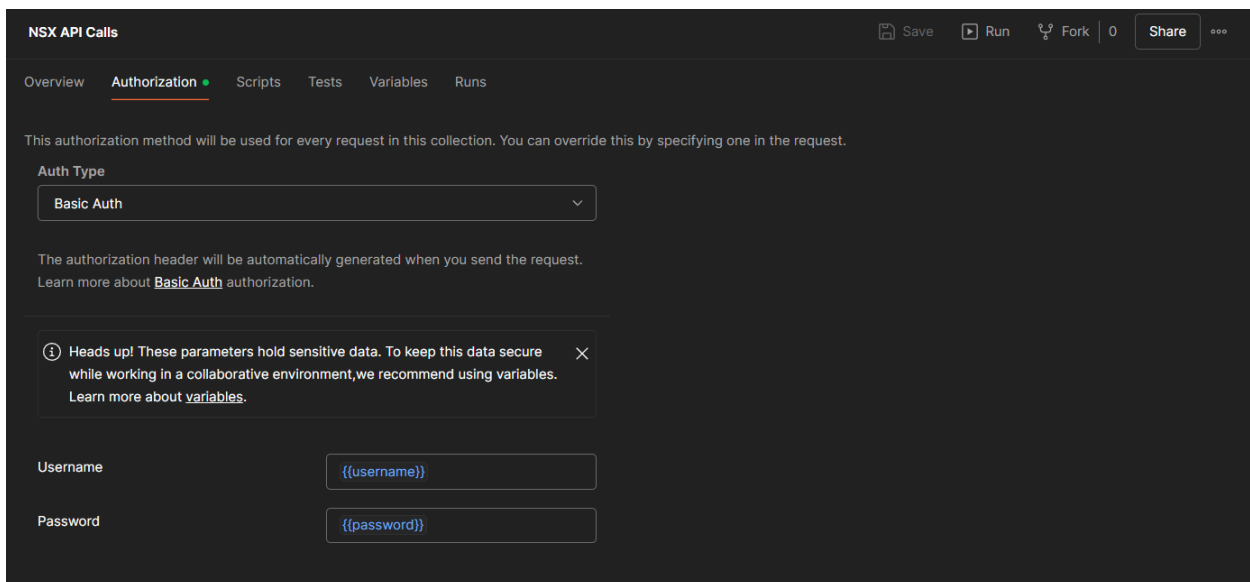
Postman

Postman is a free API platform for building and using APIs. All examples in this paper were built and tested using Postman. You can find more information [here](#)

Authentication

Most API calls require authentication. The NSX Policy API supports several forms of authentication but we will focus on a few of the most common ones that will enable us to get started with automation quickly.

The first method is using basic authentication using a username and password with sufficient rights to perform CRUD operations against the NSX Manager. In Postman, this can be done easily by setting the "Auth Type" in the "Authorization" tab of the call or collection. The example below sets the username and password using Postman environment variables `{{username}}` and `{{password}}`. This method is great when you have multiple environments and need to swap credentials quickly.



The second method is using a basic authentication in base64. This is a base64 string that is created by encoding the username and password separated by a single colon (:) into a single base64 string. To generate this in Windows, you can use the following command:

```
[convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes(username:password))
```

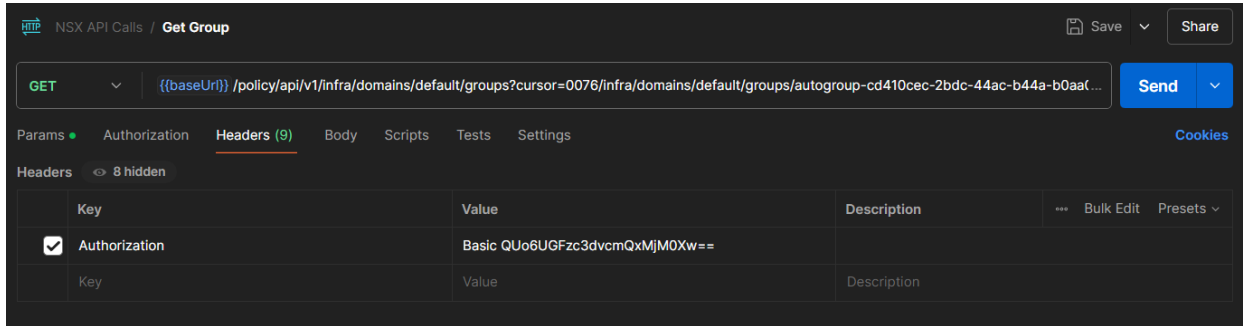
For linux you use the following command:

```
echo -n 'username:password' | base64
```

Special characters can be used with these commands without the need for escape characters (\).

Beginners Guide to Automation with vDefend Firewall

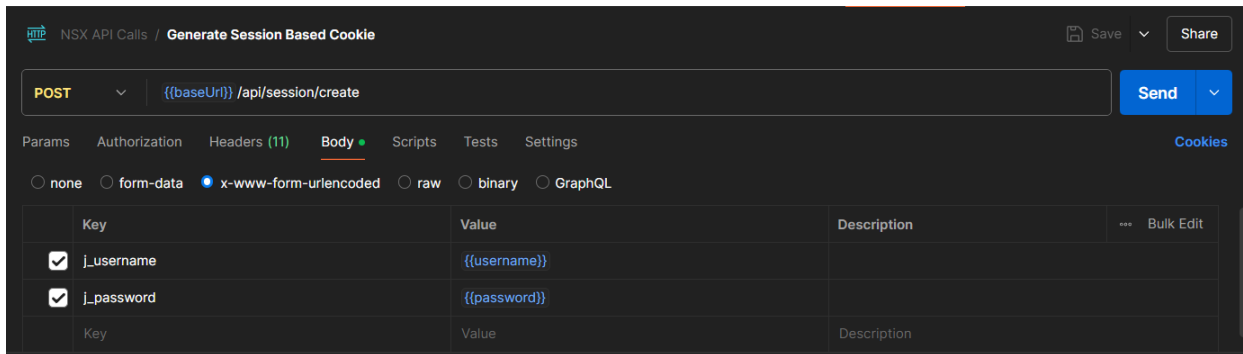
Once you have the base64 output, you set Postman Authorization to "No Auth". This is because the authentication will be handled by identifying it in the header of the call. Go to the header section of the call and define an Authentication Key with the value being "Basic <BASE64 TOKEN HERE>". An example is below:



The final method we will discuss is using session based authentication. A call to the NSX Manager with the username and password will generate a unique session cookie. Per the documentation:

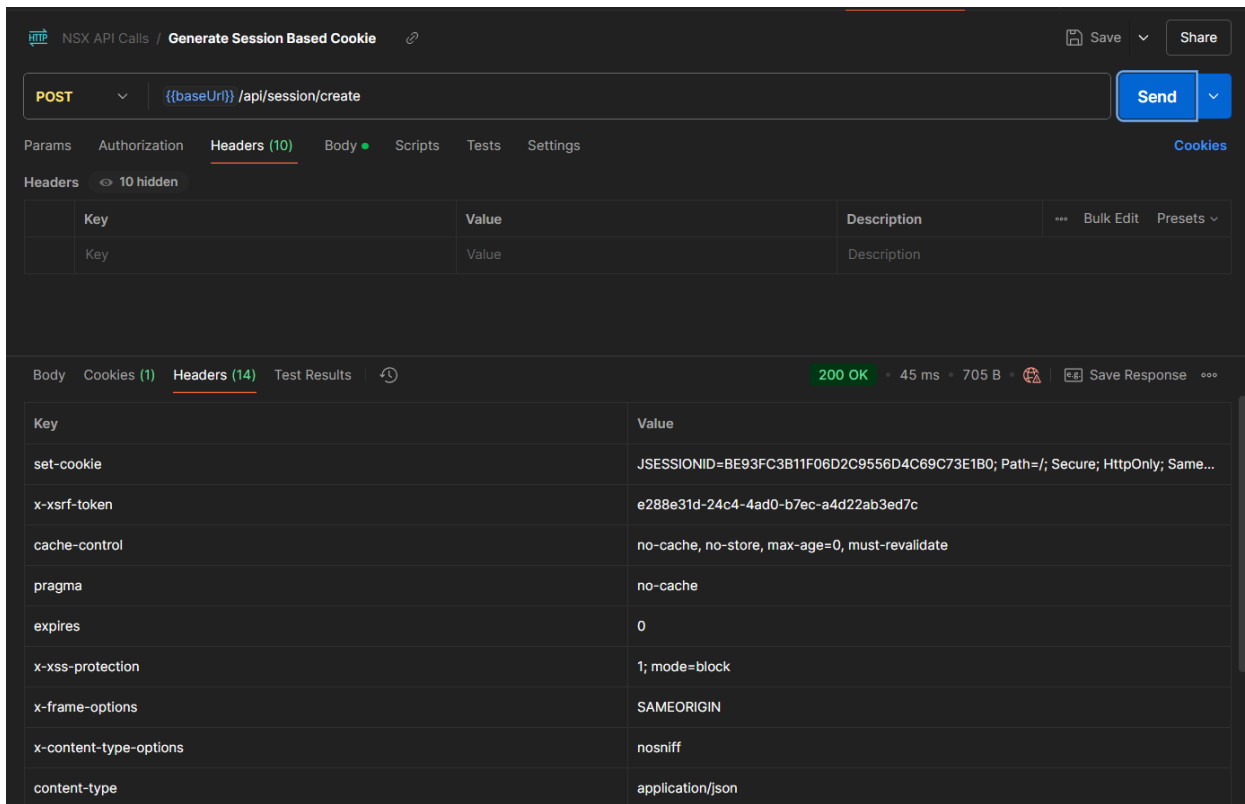
To obtain a session cookie, POST form data to the server using the application/x-www-form-urlencoded media type, with fields "j_username" and "j_password" containing the username and password separated by an ampersand. Since an ampersand is a UNIX shell metacharacter, you may need to surround the argument with single quotes.

Example:



Results from successful call:

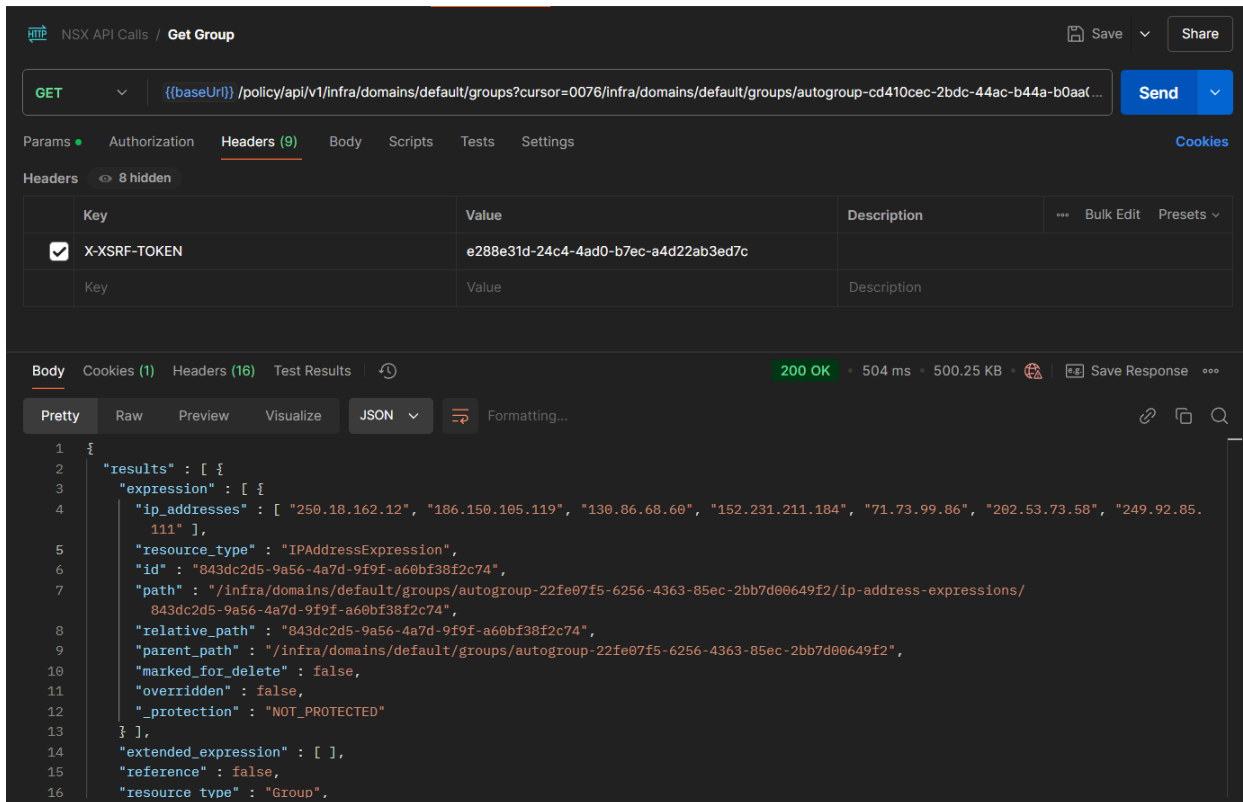
Beginners Guide to Automation with vDefend Firewall



The screenshot shows a REST client interface for a POST request to `{{baseUrl}}/api/session/create`. The response status is `200 OK` with a response time of 45 ms and a size of 705 B. The response headers are displayed in a table below.

Key	Value	Description
set-cookie	JSESSIONID=BE93FC3B11F06D2C9556D4C69C73E1B0; Path=/; Secure; HttpOnly; Same...	
x-xsrf-token	e288e31d-24c4-4ad0-b7ec-a4d22ab3ed7c	
cache-control	no-cache, no-store, max-age=0, must-revalidate	
pragma	no-cache	
expires	0	
x-xss-protection	1; mode=block	
x-frame-options	SAMEORIGIN	
x-content-type-options	nosniff	
content-type	application/json	

Now that the call was successful, there is an X-XSRF-TOKEN that can be used for API calls against the NSX Manager. Here is an example of a successful call using the XSRF Token:



Principal IDs and Certs can also be used for API calls, but that is beyond the scope of this paper. Please visit the [NSX Manager Administration](#) guide for full details on how to configure this.

Rate Limiting

When deciding your automation strategy and what tools to use, you always should consider how many requests per second you will need to execute in production. The NSX Manager has a limit in how many requests per second it can handle for safety reasons. Per the documentation [here](#), there are several limits to consider:

- 1) A per-client rate limit, in requests per second. If a client makes more requests than this limit in one second, the API server will refuse to service the API request and will return an HTTP 429 Too Many Requests Error. By default, this limit is 100 requests per second.
- 2) A per-client concurrency limit. This is the maximum number of outstanding requests that a client can have. For example, a client can open multiple connections to NSX-T and submit operations on each connection. When this limit is exceeded, the server returns a 429 Too Many Requests error to the client. By default, this limit is 40 concurrent requests.
- 3) An overall maximum number of concurrent requests. This is the maximum number of API requests that can be in process on the server. If the server is at this limit, additional requests will be refused and the HTTP error 503 Service Unavailable will be returned to the client. By default, this limit is 199 concurrent requests.

The first two limits exist to provide some level of fairness across multiple clients of NSX-T, and are intended to prevent one greedy client from preventing other clients from making API requests.

The last limit is the server's way of protecting itself against an unintentional (or intentional) denial of service attack.

While it is possible to configure these rate limits using the `/api/v1/node/services/http` API, it is not recommended. Instead, you should design your API client to gracefully deal with situations where limits are exceeded.

As stated above, it is not recommended to change the default to ensure safety and stability of the NSX Manager. API clients handle these limits gracefully with backoffs and retries.

Automation Steps

1. Gather the VM information
2. Tag the VMs
3. Create the groups with dynamic matching criteria
4. Create a custom service
5. Create a Policy
6. Create three rules inside the policy

Full Scale Automation Example

Now that we have reviewed the automation strategies and important topics like cursor, sequence numbers and authentication, let's look at the full process of identifying VMs, tagging them, creating groups, a custom service, and finally policy and a rule with automation. We will use Postman so we can focus on the calls and payloads without the need of understanding how to use a language like Python or Go.

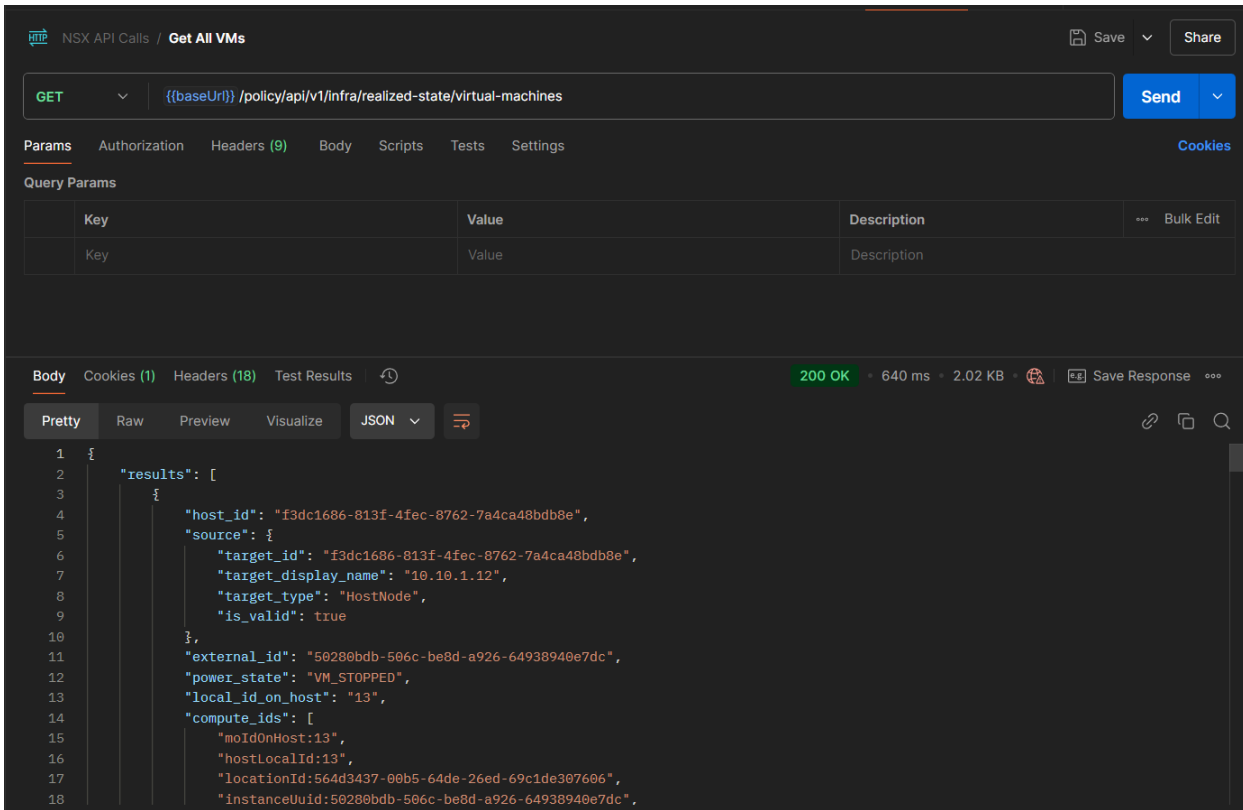
Gather VM Information:

The first call used will gather all the Virtual Machines in the NSX Manager inventory. We will use the GET method since we are receiving information. In this case, we do not have a lot of virtual machines so we won't need to worry about a cursor value. Below is the URL:

```
https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/realized-state/virtual-machine
```

Remember, you will need to include authentication information that has permissions to use the API for this call to be successful. Please see the above section on authentication for full details about how to set this up in Postman.

Once the command is run, a `'200 OK'` status code will be received indicating the call was successful. If you receive another error code, it indicates there is an error. A `'403 Forbidden'` would mean your authentication credentials are either incorrect or do not have the required permissions to execute the API call.



*Note that in my Postman examples, I use `{{baseUri}}` which is an environment variable in Postman for the NSX Manager IP

If a virtual machine already has tags, you will see it in the payload section associated with that virtual machine:



If the virtual machine does not have tags, the tag section will be omitted in the payload section of that virtual machine.

Once we have the payload, we can identify the external ID of the virtual machine we want to apply tags to.

```
{
```

Beginners Guide to Automation with vDefend Firewall

```
    <--omitted-->
    "external_id": "50281c70-8071-b9b4-9ce1-d6df54fa122e",
    "power_state": "VM_RUNNING",
    "resource_type": "VirtualMachine",
    "display_name": "Prod-Web-01",
    <--omitted-->
  }
```

An example payload for defining tags to a VM is below:

```
{
  "tags": [
    {"scope": "development", "tag": "web"}
    {"scope": "", "tag": "WINDOWS"}
  ]
}
```

*Note - This call will overwrite any tags already applied to the VM

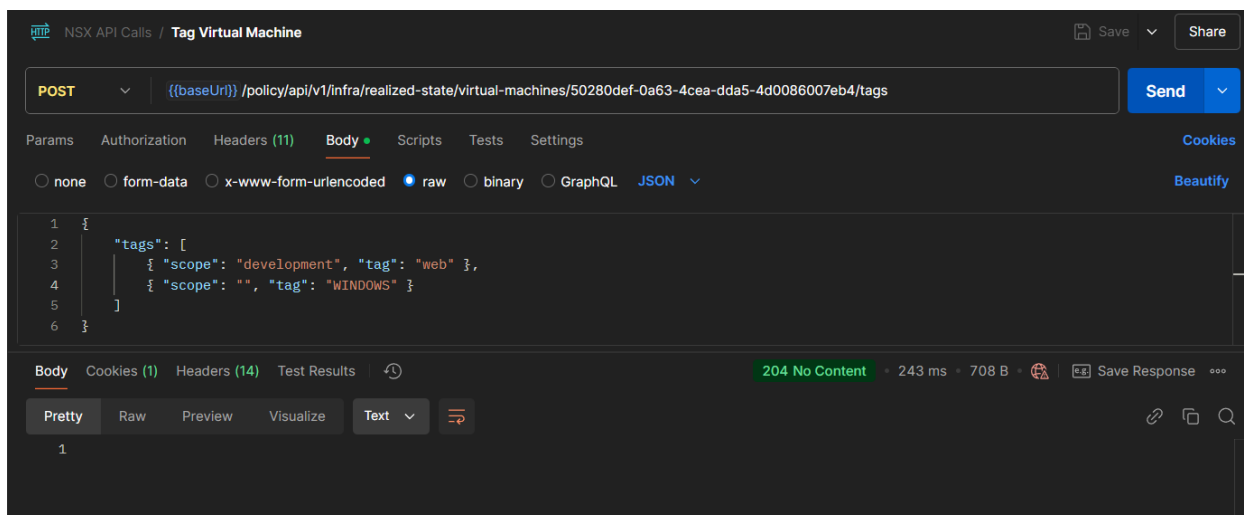
A tag name is required but a scope is optional. In this example, we will tag a virtual machine with a tag called “web” and a scope of “development”. To execute this, a POST method call must be made to the following URL:

`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/realized-state/virtual-machine/<VM_EXTERNAL_ID>/tags`

with the external ID from above, the call will look like this:

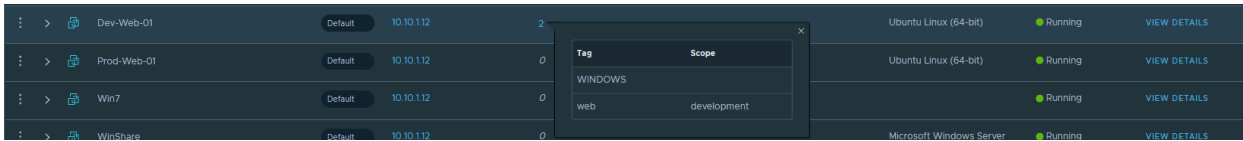
`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/realized-state/virtual-machine/50281c70-8071-b9b4-9ce1-d6df54fa122e/tags`

If the call is successful, we will receive a ‘204 No Content’ status code which means there will be no payload returned upon success.



The screenshot shows a REST client interface for a POST request to the NSX API. The URL is `{{baseUri}}/policy/api/v1/infra/realized-state/virtual-machines/50280def-0a63-4cea-dda5-4d0086007eb4/tags`. The request body is a JSON object with a "tags" array containing two objects: `{ "scope": "development", "tag": "web" }` and `{ "scope": "", "tag": "WINDOWS" }`. The response status is `204 No Content`, with a response time of 243 ms and a size of 708 B. The interface also shows tabs for Params, Authorization, Headers (11), Body, Scripts, Tests, and Settings. The Body tab is active, showing the JSON payload. The response is also shown in the bottom section, with a status of 204 No Content and a response time of 243 ms.

We can confirm via API or the NSX Manager that the virtual machine is now tagged with the two we defined in the payload:



We would repeat this process for every virtual machine we want to tag. The tags and scope can change to meet our requirements. For example, the Prod-Web-01 virtual machine will be tagged with the following:

```
{
  "tags": [
    {"scope": "production", "tag": "web"},
    {"scope": "", "tag": "WINDOWS"}
  ]
}
```

The payload format and requirements are defined in the API [documentation](#).

Groups

Groups are objects that contain a collection of one or more static or dynamic members such as IP addresses or VMs. Groups are used in the source, destination or “Applied-To” field of firewall rules. Three groups will be created with dynamic matching criteria. The tags will be used as the matching criteria to associate the VMs to the groups we create. Starting with a web group, the following payload will be used:

```
{
  "description": "WebGroup",
  "display_name": "Web Group",
  "expression": [
    {
      "member_type": "VirtualMachine",
      "value": "web",
      "key": "Tag",
      "operator": "EQUALS",
      "resource_type": "Condition"
    }
  ]
}
```

The following URL is used for the call:

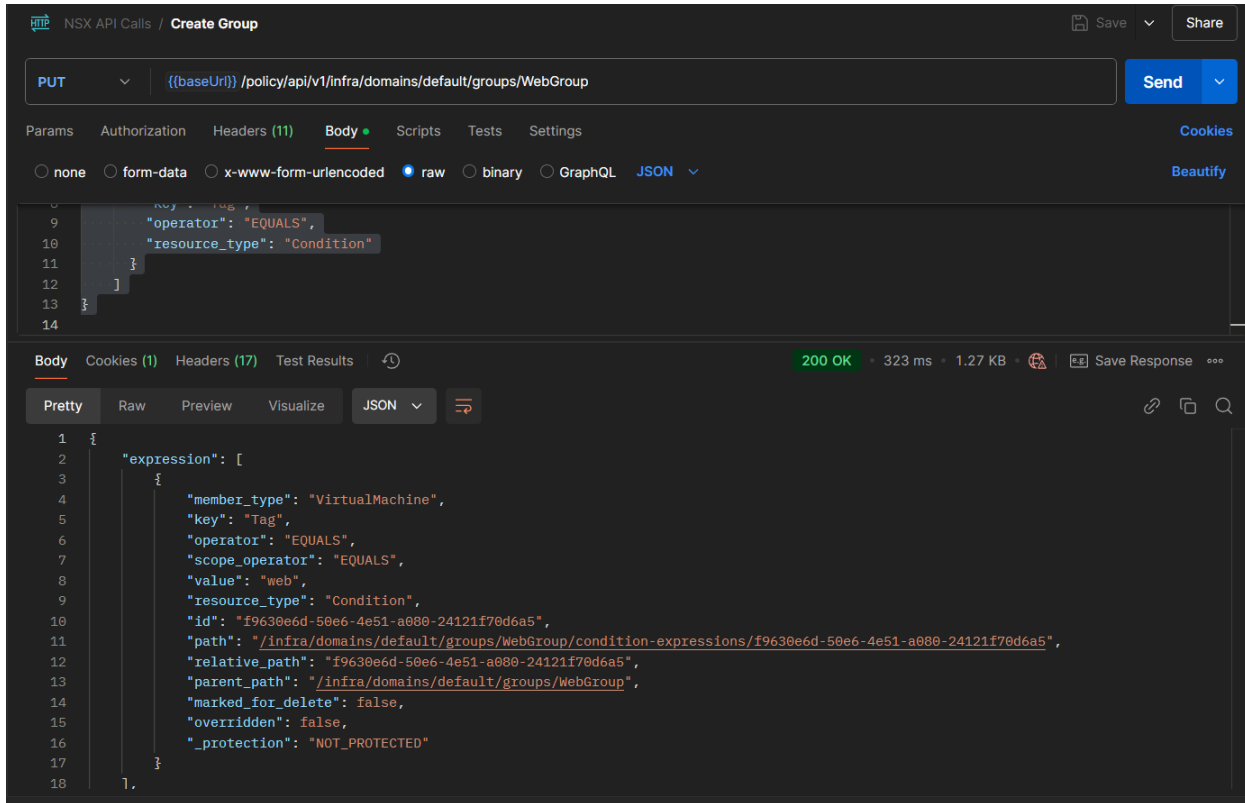
```
https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/groups/<ID>
```

*note the domain of “default” used above is the default domain used when not using projects. If projects are being used, the call would be different.

Beginners Guide to Automation with vDefend Firewall

We will set the ID of the group as WebGroup by using that name in the URL like below:

`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/groups/WebGroup`



'200 OK' indicates the call was successful and we get a payload containing all of the group information.

The follow are the URLs and payloads for creating the production and development groups to match on the scope of 'production' and 'development' respectively.

`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/groups/production`

```
{
  "description": "Production",
  "display_name": "Production",
  "expression": [
    {
      "member_type": "VirtualMachine",
      "key": "Tag",
      "operator": "EQUALS",
      "scope_operator": "EQUALS",
      "value": "production",
      "resource_type": "Condition"
    }
  ]
}
```



```
}  
]  
}
```

View Members | Production

To edit below content, click [EDIT](#)

Group Type: Generic

Effective Members **Group Definition**

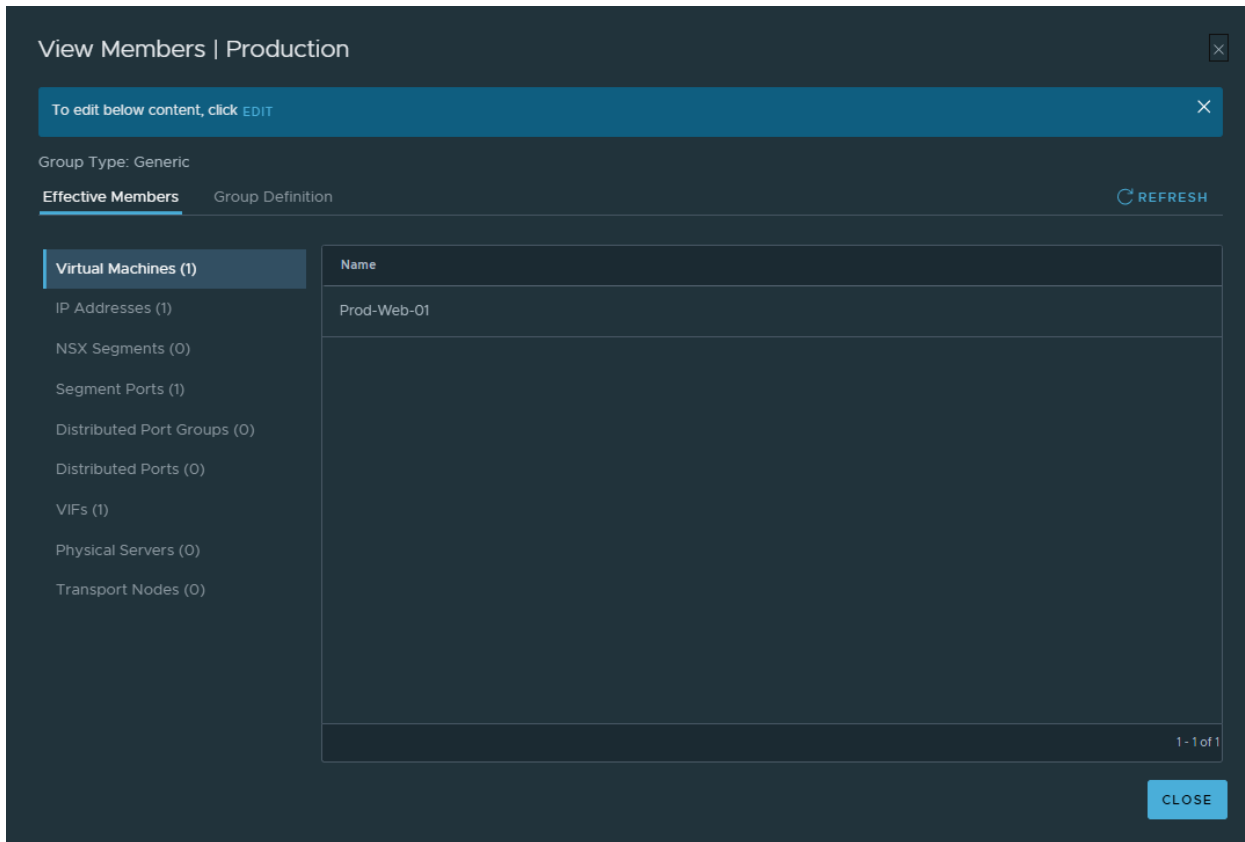
Membership Cri_ (1)

- Members (0)
- IP Addresses (0)
- MAC Addresses (0)
- AD Groups (0)

▼ Criterion 1

Virtual Machine	Tag	Equals	Scope	Equals	production
-----------------	-----	--------	-------	--------	-------------------

[CLOSE](#)



https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/groups/development

```
{
  "description": "Development",
  "display_name": "Development ",
  "expression": [
    {
      "member_type": "VirtualMachine",
      "key": "Tag",
      "operator": "EQUALS",
      "scope_operator": "EQUALS",
      "value": "development",
      "resource_type": "Condition"
    }
  ]
}
```

View Members | Development ×

To edit below content, click [EDIT](#) ×

Group Type: Generic

Effective Members **Group Definition**

Membership Cri... (1)

Members (0)

IP Addresses (0)

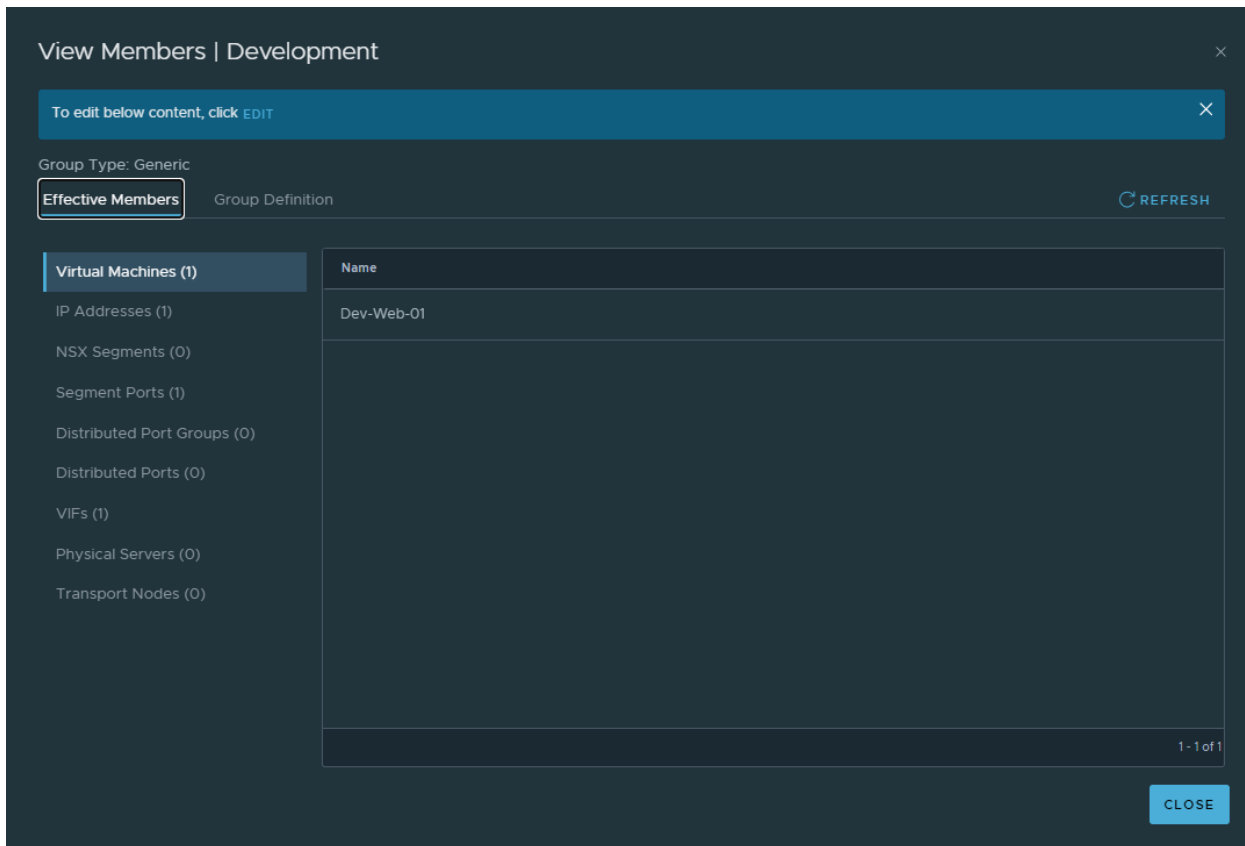
MAC Addresses (0)

AD Groups (0)

▼ Criterion 1

Virtual Machine	Tag	Equals	Scope	Equals	developme...
-----------------	-----	--------	-------	--------	------------------------------

CLOSE



Custom Services

While the default services can be used for the most popular ports, the policy API gives you the option to create your own custom named services with one or many ports associated to them. The payload for the custom service:

```
{
  "description": "Custom Service",
  "display_name": "TCP.1234",
  "service_entries": [
    {
      "resource_type": "L4PortSetServiceEntry",
      "display_name": "TCP.1234",
      "destination_ports": [
        "1234"
      ],
      "l4_protocol": "TCP"
    }
  ]
}
```

Beginners Guide to Automation with vDefend Firewall

For full information on how to create UDP service entries, or any other type, please see the [API documentation](#).

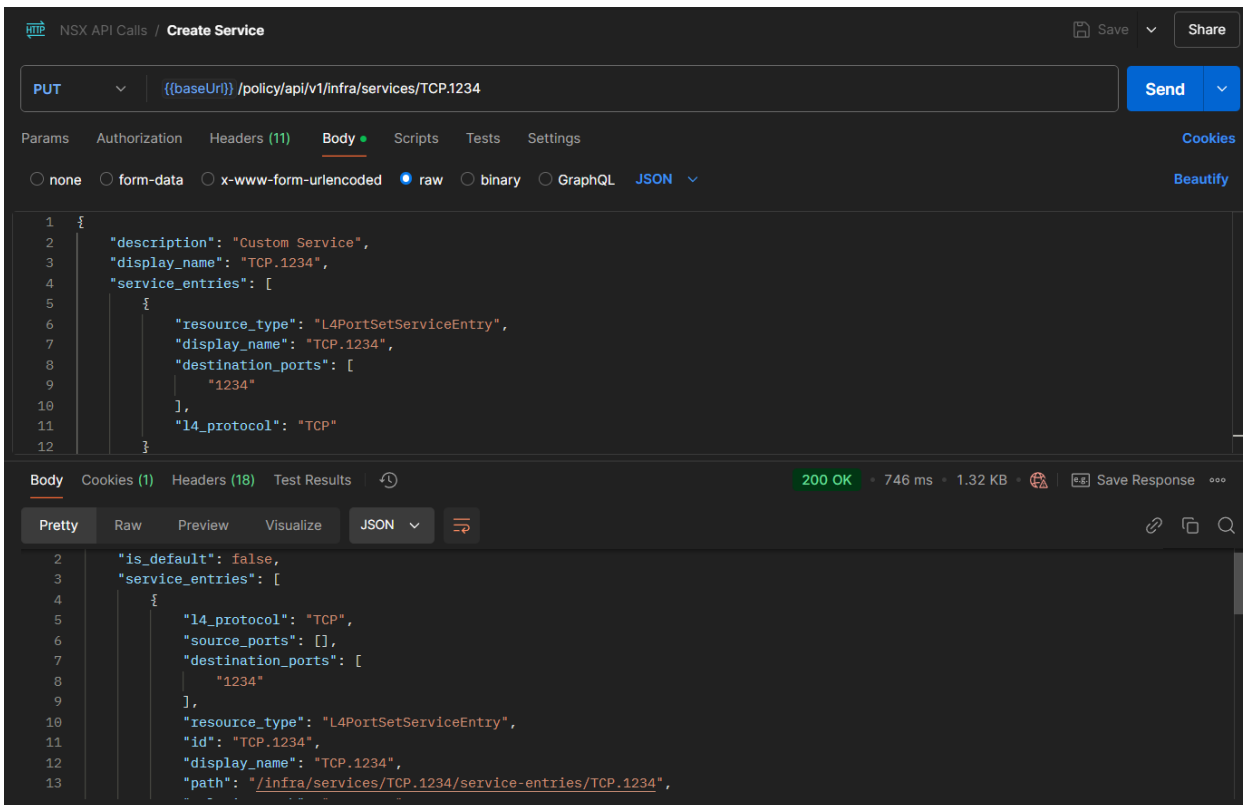
The URL for the call:

`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/infra/services/<Service_Id>`

with the service ID we want to use, it will be:

https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/infra/services/TCP.1234

If the call is successful, a '200 OK' status code is received:



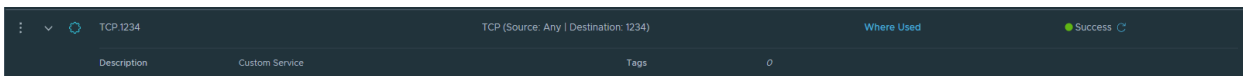
The screenshot shows an API client interface for a PUT request to the endpoint `[[baseUri]] /policy/api/v1/infra/services/TCP.1234`. The request body is a JSON object:

```
1 {
2   "description": "Custom Service",
3   "display_name": "TCP.1234",
4   "service_entries": [
5     {
6       "resource_type": "L4PortSetServiceEntry",
7       "display_name": "TCP.1234",
8       "destination_ports": [
9         "1234"
10      ],
11      "l4_protocol": "TCP"
12    }
13  ]
14 }
```

The response is a 200 OK status code, received in 746 ms and 1.32 KB. The response body is a JSON object:

```
2 "is_default": false,
3 "service_entries": [
4   {
5     "l4_protocol": "TCP",
6     "source_ports": [],
7     "destination_ports": [
8       "1234"
9     ],
10    "resource_type": "L4PortSetServiceEntry",
11    "id": "TCP.1234",
12    "display_name": "TCP.1234",
13    "path": "/infra/services/TCP.1234/service-entries/TCP.1234",
14  }
15 ]
```

The new service is seen in the GUI:



The screenshot shows the vDefend GUI interface. The top bar displays the service name 'TCP.1234' and its details 'TCP (Source: Any | Destination: 1234)'. The status is 'Success'. Below the top bar, there is a table with columns for 'Description', 'Custom Service', 'Tags', and '0'.

Policy

All the components that are needed have been created for the rules we want to make. Using the API, a combination of pre-made groups and services will be used alongside the new ones created above.

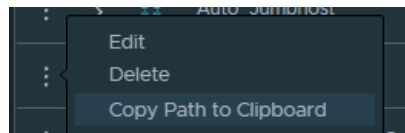
Name	Rule 1
Source	RFC1918
Destination	WebGroup
Services	HTTP/HTTPS
Action	Allow
Scope	WebGroup

Name	Rule 2
Source	RFC1918
Destination	Development
Services	TCP.1234
Action	Allow
Scope	Development

Name	Rule 3
Source	RFC1918
Destination	Production
Services	TCP.1234
Action	Deny
Scope	Production

RFC1918

The RFC1918 group has already been created and the path for the group can be found using the API or GUI. When using the GUI, the path can be found by clicking on the three dots next the RFC1918 group in the NSX Manager.



The call to get the path is the same get call we would use to gather any other information about a group:

```
https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domain/default/groups/RFC1918
```

Below is the payload for the first rule:

```
{
  <--Omitted-->
  "resource_type": "Group",
  "id": "RFC1918",
  "display_name": "RFC1918",
  "path": "/infra/domains/default/groups/RFC1918",
  <--Omitted-->
}
```

The path is needed for creating rules in the NSX Manager. For groups, it will be in the format of:

```
/infra/domains/default/groups/<GROUP_ID>
```

First, we need to create an empty policy for our new rules to live in.

Policy with Rule(s)

When creating policy, rules can also be created within the same payload. For the purposes of this demo, they have been separated into multiple calls. For details of how the payload looks, please refer to the [API documentation](#)

```
{  
  "description": "Automation Rules",  
  "display_name": "Automation Policy",  
  "category": "Application"  
}
```

The URL for the PUT action is:

```
https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/security-policies/<Policy_ID>
```

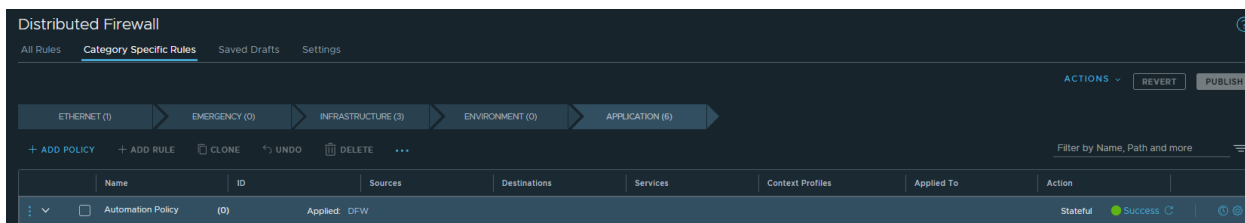
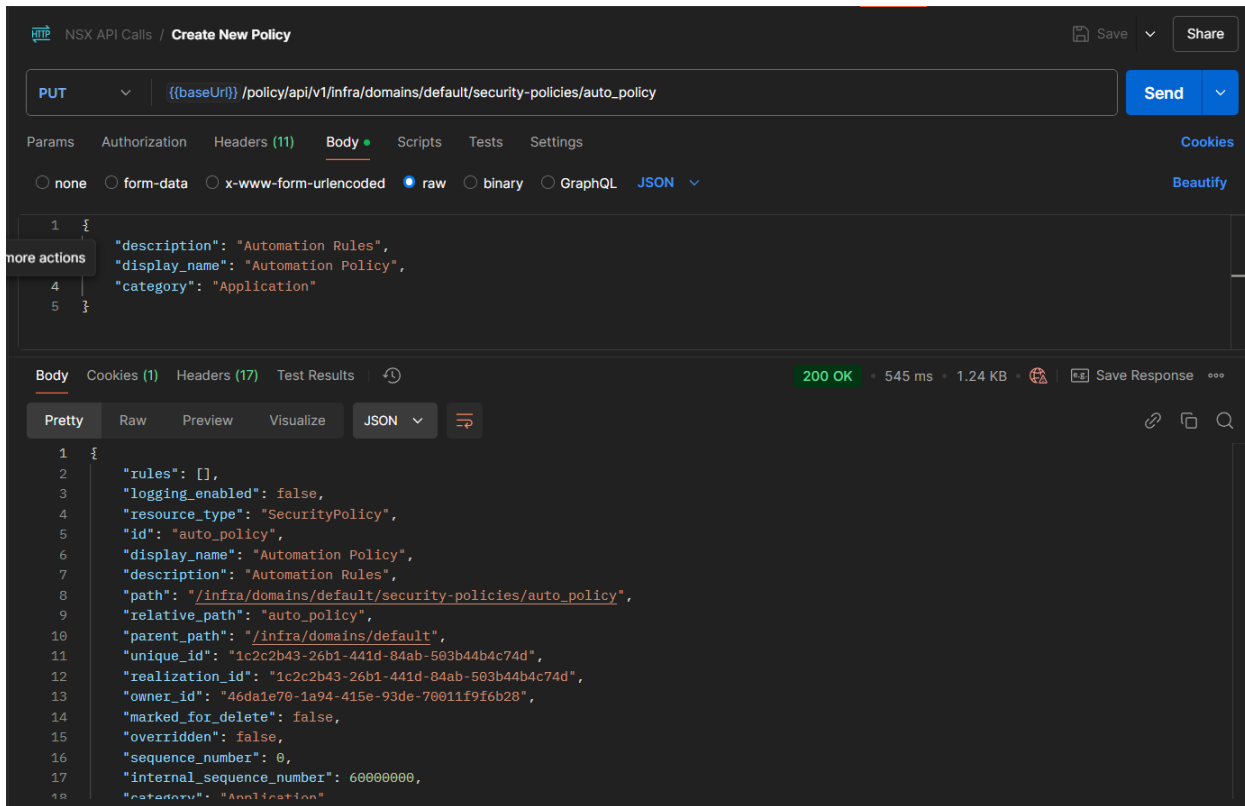
The policy will be named “auto_policy”:

```
https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/security-policies/auto_policy
```

Policies can be placed in one of the following categories in the NSX Manager:

- Ethernet
- Emergency
- Infrastructure
- Environmental
- Application

If successful, a '200 OK' is received with all the information about the new section. Rules can be included in the payload for creation, but in this example we are separating out the steps.



Now that there is an empty policy in the Application category, rules can be created in this area. Rule 1's URL and payload is included below:

`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/security-policies/auto_policy/rules/rule1`

```

{
  "action" : "ALLOW",
  "description" : "Test Group with Automation",
  "source_groups" : [
    "/infra/domains/default/groups/RFC1918"
  ],
  "destination_groups" : [
    "/infra/domains/default/groups/WebGroup"
  ],
  "services" : [

```

Beginners Guide to Automation with vDefend Firewall

```
    "/infra/services/HTTP",  
    "/infra/services/HTTPS"  
  ],  
  "scope": [  
    "/infra/domains/default/groups/WebGroup"  
  ]  
}
```

If the call is successful, a '200 OK' is received:

The screenshot shows the NSX API Client interface for a PUT request. The URL is `{{baseUri}}/policy/api/v1/infra/domains/default/security-policies/auto_policy/rules/rule1`. The request body is a JSON object: `{ "action": "ALLOW", "description": "Rule 1 with Automation", "source_groups": ["/infra/domains/default/groups/RFC1918"], "destination_groups": [] }`. The response is a 200 OK with a status of 166 ms and 1.29 KB. The response body is displayed in JSON format: `{ "action": "ALLOW", "resource_type": "Rule", "id": "rule1", "display_name": "rule1", "description": "Rule 1 with Automation", "path": "/infra/domains/default/security-policies/auto_policy/rules/rule1", "relative_path": "rule1", "parent_path": "/infra/domains/default/security-policies/auto_policy", "unique_id": "00000000-0000-0000-0000-0000000011242", "realization_id": "00000000-0000-0000-0000-0000000011242", "owner_id": "46da1e70-1a94-415e-93de-70011f9f6b28", "marked_for_delete": true, "overridden": false, "rule_id": 11242, "sequence_number": 0, "sources_excluded": false, "destinations_excluded": false }`

The screenshot shows the Distributed Firewall console. The 'Category Specific Rules' tab is active, showing a list of rules. The rule 'rule1' is highlighted, showing its configuration: Name: rule1, ID: 11241, Sources: RFC1918, Destinations: Web Group, Services: HTTP, HTTPS, Context Profiles: None, Applied To: Web Group, Action: Allow. The rule is in a 'Success' state.

The URL and payload for the other two rules are listed below:

`https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/security-policies/auto_policy/rules/rule2`

Beginners Guide to Automation with vDefend Firewall

```
{
  "action" : "ALLOW",
  "description" : "Rule 2 with Automation",
  "source_groups" : [
    "/infra/domains/default/groups/RFC1918"
  ],
  "destination_groups" : [
    "/infra/domains/default/groups/development"
  ],
  "services" : [
    "/infra/services/TCP.1234"
  ],
  "scope": [
    "/infra/domains/default/groups/development"
  ]
}
```

The screenshot shows the NSX API Client interface for a PUT request. The URL is `{{baseUri}}/policy/api/v1/infra/domains/default/security-policies/auto_policy/rules/rule2`. The request body is raw JSON, and the response is also raw JSON. The response status is 200 OK, with a response time of 180 ms and a size of 1.29 KB. The response body is displayed in a Pretty JSON format, showing the details of the created rule.

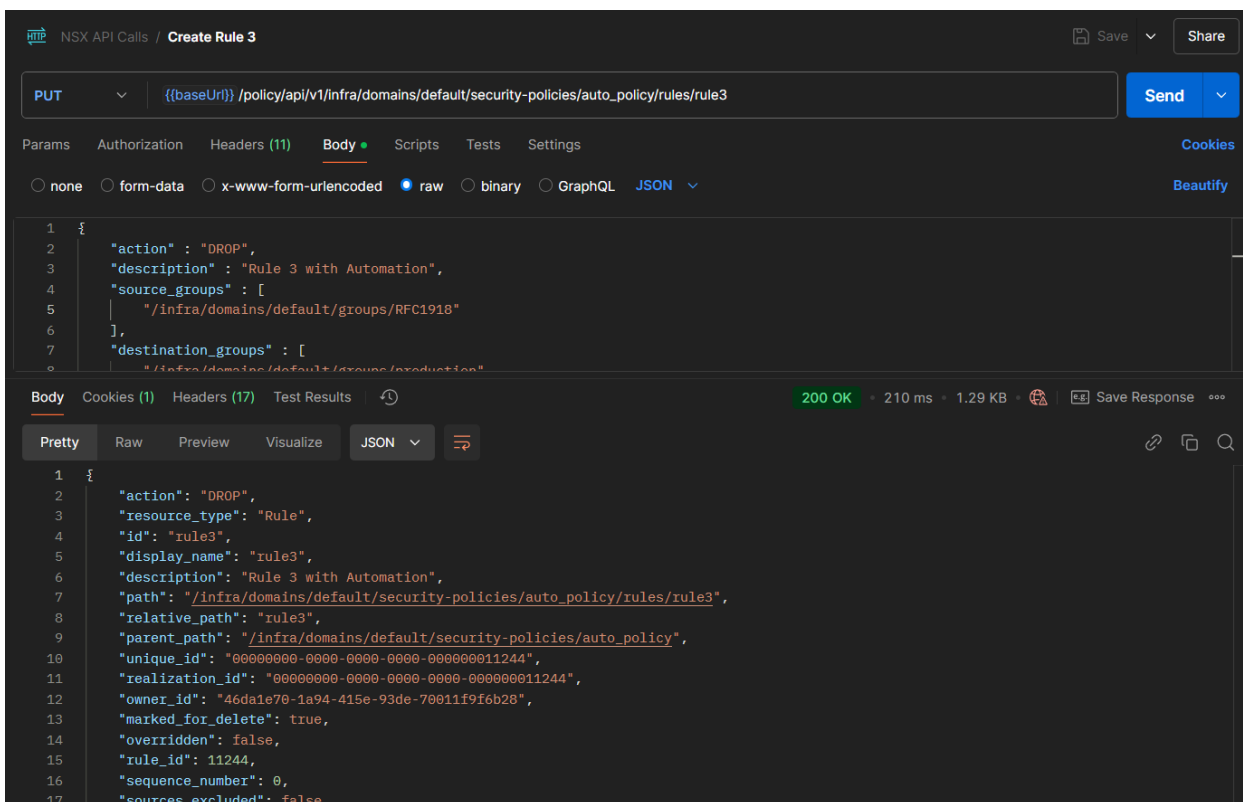
```
1 {
2   "action": "ALLOW",
3   "description": "Rule 2 with Automation",
4   "source_groups": [
5     "/infra/domains/default/groups/RFC1918"
6   ],
7   "destination_groups": [
8     "/infra/domains/default/groups/development"
9   ]
10 }
11
12 "action": "ALLOW",
13 "resource_type": "Rule",
14 "id": "rule2",
15 "display_name": "rule2",
16 "description": "Rule 2 with Automation",
17 "path": "/infra/domains/default/security-policies/auto_policy/rules/rule2",
18 "relative_path": "rule2",
19 "parent_path": "/infra/domains/default/security-policies/auto_policy",
20 "unique_id": "00000000-0000-0000-0000-000000011243",
21 "realization_id": "00000000-0000-0000-0000-000000011243",
22 "owner_id": "46da1e70-1a94-415e-93de-70011f9f6b28",
23 "marked_for_delete": true,
24 "overridden": false,
25 "rule_id": 11243,
26 "sequence_number": 0,
27 "sources_excluded": false.
```

https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/domains/default/security-policies/auto_policy/rules/rule3

```
{
  "action" : "DROP",
```

Beginners Guide to Automation with vDefend Firewall

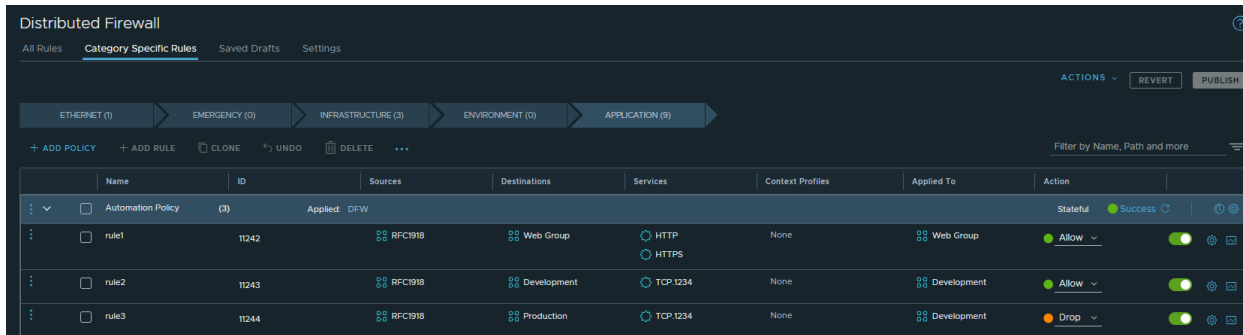
```
"description" : "Rule 3 with Automation",
"source_groups" : [
  "/infra/domains/default/groups/RFC1918"
],
"destination_groups" : [
  "/infra/domains/default/groups/production"
],
"services" : [
  "/infra/services/TCP.1234"
],
"scope": [
  "/infra/domains/default/groups/development"
]
}
```



The screenshot displays the NSX API Calls interface for a PUT request to create a rule. The URL is `{{baseUri}}/policy/api/v1/infra/domains/default/security-policies/auto_policy/rules/rule3`. The request body is shown in raw JSON format, matching the code block above. The response is a 200 OK status with a response time of 210 ms and a size of 1.29 KB. The response body is shown in pretty-printed JSON format, containing detailed metadata for the created rule.

```
1 {
2   "action": "DROP",
3   "resource_type": "Rule",
4   "id": "rule3",
5   "display_name": "rule3",
6   "description": "Rule 3 with Automation",
7   "path": "/infra/domains/default/security-policies/auto_policy/rules/rule3",
8   "relative_path": "rule3",
9   "parent_path": "/infra/domains/default/security-policies/auto_policy",
10  "unique_id": "00000000-0000-0000-0000-0000000011244",
11  "realization_id": "00000000-0000-0000-0000-0000000011244",
12  "owner_id": "46da1e70-1a94-415e-93de-70011f9f6b28",
13  "marked_for_delete": true,
14  "overridden": false,
15  "rule_id": 11244,
16  "sequence_number": 0,
17  "sources_excluded": false,
```

All the rules have been created in the NSX Manager



Hierarchical API Example:

The example above can be replicated using the hierarchical API with the exception of tagging the Virtual Machines. The Virtual Machines can be tagged using the singular calls above or using the UI before or after running this payload in Postman. To use the hierarchical API call, use the following URI with a method of PATCH:

https://<NSXMANAGER_IP_OR_FQDN>/policy/api/v1/infra

```
{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildService",
      "Service": {
        "description": "Custom Service",
        "display_name": "TCP.1234",
        "id": "TCP.1234",
        "resource_type": "Service",
        "service_entries": [
          {
            "resource_type": "L4PortSetServiceEntry",
            "display_name": "TCP.1234",
            "destination_ports": [
              "1234"
            ],
            "l4_protocol": "TCP"
          }
        ]
      }
    },
    {
      "resource_type": "ChildDomain",

```

```
"marked_for_delete": false,
"Domain": {
  "id": "default",
  "resource_type": "Domain",
  "marked_for_delete": false,
  "children": [
    {
      "resource_type": "ChildGroup",
      "Group": {
        "id": "WebGroup",
        "display_name": "Web Group",
        "resource_type": "Group",
        "expression": [
          {
            "member_type": "VirtualMachine",
            "value": "web",
            "key": "Tag",
            "operator": "EQUALS",
            "resource_type": "Condition"
          }
        ]
      }
    }
  ],
},
{
  "resource_type": "ChildGroup",
  "Group": {
    "id": "Production",
    "display_name": "Production",
    "resource_type": "Group",
    "expression": [
      {
        "member_type": "VirtualMachine",
        "value": "production",
        "key": "Tag",
        "operator": "EQUALS",
        "resource_type": "Condition"
      }
    ]
  }
},
```

```
{
  "resource_type": "ChildGroup",
  "Group": {
    "id": "Development",
    "display_name": "Development",
    "resource_type": "Group",
    "expression": [
      {
        "member_type": "VirtualMachine",
        "value": "development",
        "key": "Tag",
        "operator": "EQUALS",
        "resource_type": "Condition"
      }
    ]
  }
},
{
  "resource_type": "ChildSecurityPolicy",
  "SecurityPolicy": {
    "resource_type": "SecurityPolicy",
    "display_name": "Automation Rules",
    "id": "automation_policy",
    "marked_for_delete": false,
    "tcp_strict": true,
    "stateful": true,
    "locked": false,
    "category": "Application",
    "sequence_number": 2,
    "children": [
      {
        "resource_type": "ChildRule",
        "marked_for_delete": false,
        "Rule": {
          "display_name": "Rule 1",
          "id": "rule_1",
          "resource_type": "Rule",
          "marked_for_delete": false,
          "source_groups": [
```

```
"/infra/domains/default/groups/RFC1918"
```

```
    ],
    "sequence_number": 10,
    "destination_groups": [
"/infra/domains/default/groups/WebGroup"
    ],
    "services": [
        "/infra/services/HTTPS",
        "/infra/services/HTTP"
    ],
    "profiles": [
        "ANY"
    ],
    "scope": [
        "/infra/domains/default/groups/RFC1918",
        "/infra/domains/default/groups/WebGroup"
    ],
    "action": "ALLOW",
    "direction": "IN_OUT",
    "logged": false,
    "disabled": false,
    "notes": "",
    "tag": "",
    "ip_protocol": "IPV4_IPV6"
}
},
{
"resource_type": "ChildRule",
"marked_for_delete": false,
"Rule": {
    "display_name": "Rule 2",
    "id": "rule_2",
    "resource_type": "Rule",
    "marked_for_delete": false,
    "source_groups": [
        "/infra/domains/default/groups/RFC1918"
    ],
    "sequence_number": 20,
    "destination_groups": [
        "/infra/domains/default/groups/Development"
    ],
    ],
```



```

        "services": [
            "/infra/services/TCP.1234"
        ],
        "profiles": [
            "ANY"
        ],
        "scope": [
            "/infra/domains/default/groups/RFC1918",
            "/infra/domains/default/groups/Development"
        ],
        "action": "ALLOW",
        "direction": "IN_OUT",
        "logged": false,
        "disabled": false,
        "notes": "",
        "tag": "",
        "ip_protocol": "IPV4_IPV6"
    }
},
{
    "resource_type": "ChildRule",
    "marked_for_delete": false,
    "Rule": {
        "display_name": "Rule 3",
        "id": "rule_3",
        "resource_type": "Rule",
        "marked_for_delete": false,
        "source_groups": [
            "/infra/domains/default/groups/RFC1918"
        ],
        "sequence_number": 30,
        "destination_groups": [
            "/infra/domains/default/groups/Production"
        ],
        "services": [
            "/infra/services/TCP.1234"
        ],
        "profiles": [
            "ANY"
        ],
    },

```

```
        "scope": [
            "/infra/domains/default/groups/RFC1918",
            "/infra/domains/default/groups/Production"
        ],
        "action": "DROP",
        "direction": "IN_OUT",
        "logged": false,
        "disabled": false,
        "notes": "",
        "tag": "",
        "ip_protocol": "IPV4_IPV6"
    }
}
]
}
]
}
]
}
]
}
]
```

The full payload is available in this [public repo](#).

Summary

In this paper we discussed the different automation strategies, languages, and products that can be used to automate vDefend policy in the NSX Manager. Using the examples above coupled with a language (Python, Go, Javascript, etc) or product (Aria, Terraform, etc), there are numerous possibilities available to quickly deploy new objects and security policy with vDefend. Automation is essential in order for IT operations teams to keep up with infrastructure security demands. This paper should serve as a simple starting point to enable you to create the needed components to keep up.

Helpful Links and Coding Examples:

[API Documentation](#)

[NSX Sample Code](#)

[Hierarchical API Postman Example Payload](#)

[Terraform](#)

[Ansible](#)

[PowerCLI](#)

[GoLang SDK](#)

[Python SDK](#)

