BeyondTrust

DevOps Secrets Safe 20.3
Getting Started

TC:9/9/2020

Table of Contents

Getting Started with DevOps Secrets Safe

Before proceeding with this section, please ensure a new instance of DevOps Secrets Safe (DSS) is running and that the DSS Command Line Interface (CLI) is configured to communicate with it.

For more information about how to configure the DevOps Secrets Safe CLI, please see Install the DevOps Secrets Safe CLI.

Initialize DevOps Secrets Safe

1. Initialize DevOps Secrets Safe:

   ```
   $ ssrun init
   ```

   Set the desired password for the root user account in the DSS instance when prompted. The password must be at least 10 characters long. A successful call to initialize returns the master key for this DSS instance. Save this key to a file.

Note: *The remainder of this guide assumes that the root account password for the DevOps Secrets Safe instance has been set to* ***rootpassword*** *and that the master key has been saved to a file called* ***master.txt****.*

2. Unseal DevOps Secrets Safe:

   ```
   $ ssrun unseal -f master.txt
   ```

   All CLI commands aside from **Initialize** and **Unseal** are unavailable until the instance is unsealed. This command puts the DevOps Secrets Safe application into a state where secrets may be saved and retrieved.

3. Log in to DevOps Secrets Safe as root:

   ```
   $ ssrun login -u root -p rootpassword
   ```

Manage Users

Before starting this section, ensure you have initialized, unsealed, and logged into DevOps Secrets Safe as root.

1. Create a new user:

   ```
   $ ssrun user create -n NewUser -p NewUserPassword
   ```

Note: *Passwords must be 10 characters in length.*

2. View the list of users:

```
$ ssrun user get -v
```

> ***Note:*** *The principal discovery mechanism accepts any subset of the URI* ***{identity_provider}/{principal_type}/{principal_name}/{principal_extension_data}****. Therefore, the URI above returns all internal users. Additionally, the (optional)* ***-v*** *flag can be used to get a full listing of principals or principal containers attributes. Otherwise, a slim view of each principal or principal container is returned.*

3. Create a secret:

```
$ echo -n "I love my test content" | ssrun secret create testsecret:mytestsecret
```

> ***Note:*** *Whenever you reference a secret, the URI must be in the format* ***{scopePath}:{secretName}****. For example,* ***path/to/secrets:secretName****.*

> ***Note:*** *The echo line may only be performed in bash and similar shells.*

4. Authorize the new user to read the secret:

   The create-authorization command accepts the following arguments:

```
$ ssrun authorization create -p principal/internal/user/NewUser -o read -a allow
secret/testsecret:mytestsecret
```

   - **-p**: (Required). URI of the principal the access control is being applied to.
     - A user's URI can be derived using the principal discovery mechanism detailed in step 2.
   - **-o**: (Optional). Operations authorization applies to.
     - Options are **create** | **read** | **update** | **delete** .
   - **-a**: (Optional). Set to allow to grant authorization or deny to revoke.

5. Log in as the new user:

```
$ ssrun login -u NewUser -p NewUserPassword
```

6. Read the secret:

```
$ ssrun secret get testsecret:mytestsecret
```

7. Log in as root again:

```
$ ssrun login -u root -p rootpassword
```

8. Delete the new user:

```
$ ssrun user delete -n NewUser
```

For more information, please see the following:

- For information on how to initialize DSS, "Initialize DevOps Secrets Safe" on page 5
- For information on managing secrets, "Manage Secrets and Scopes" on page 8

Manage Applications

Before starting this section, ensure you have initialized, unsealed, and logged into DevOps Secrets Safe as root.

1. Create a new application:

```
$ ssrun application create -n NewApplication
```

Note: *Upon creation, an API key is returned. This will be used in any subsequent log in.*

2. View the list of applications:

```
$ ssrun application get -v
```

Note: *The principal discovery mechanism in the API accepts any subset of the URI* ***{identity_provider}/{principal_type}/{principal_name}/{principal_extension_data}****. Therefore, the command above returns all internal applications. Additionally, the (optional)* ***-v*** *flag can be used to get a full listing of principals or principal containers attributes. Otherwise, a slim view of each principal or principal container is returned.*

3. Create a secret:

```
$ echo -n "I love my test content" | ssrun secret create testsecret:mytestsecret
```

Note: *Whenever you reference a secret, the URI must be in the format* ***{scopePath}:{secretName}****. For example,* ***path/to/secrets:secretName****.*

Note: *The echo line may only be performed in bash and similar shells.*

4. Authorize the new application to read the secret:

```
$ ssrun authorization create -p principal/internal/application/NewApplication -o read -a allow
secret/testsecret:mytestsecret
```

The **authorization** command accepts the following arguments:

- **-p**: (Required). URI of the principal the access control is being applied to.
  - An applications URI can be derived using the principal discovery mechanism detailed in step 2.
- **-o**: (Optional). Operations authorization applies to.
  - Options are create | **read** | **update** | **delete** | **grant**.
- **-a**: (Required). Set to allow to grant authorization or deny to revoke.
  - Options are **allow** | **deny**.

5. Log in as the new application:

```
$ ssrun login -a NewApplication -k 2a098f21-0b11-4918-b705-7752588d5d8c
```

***Note:** The API key **-k** comes from what was returned when the application was created in step 1.*

6. Read the secret:

```
$ ssrun secret get testsecret:mytestsecret
```

7. Log in as root again:

```
$ ssrun login -u root -p rootpassword
```

8. Delete the new application:

```
$ ssrun application delete -n NewApplication
```

***Note:** The name associated with an application can be determined via the list applications command as detailed in step 2.*

For more information, please see the following:

- For information about how to log into DSS as root, "Initialize DevOps Secrets Safe" on page 5
- For information on managing secrets, "Manage Secrets and Scopes" on page 8

Manage Secrets and Scopes

Before starting this section, ensure you have initialized, unsealed, and logged into DevOps Secrets Safe as root.

The next example assumes there are two files, **myTestSecretData1.txt** and **myTestSecretData2.txt**, containing data you want to store as secrets.

1. Create two secrets:

```
$ ssrun secret create -f myTestSecretData1.txt path/to/my/secrets:mytestsecret1
```

```
$ ssrun secret create -f myTestSecretData2.txt path/to/my/secrets:mytestsecret2
```

> ***Note:*** *Whenever you reference a secret, the URI must be in the format* ***{scopePath}:{secretName}****. For example,* ***path/of/scope:secretName****.*

2. Retrieve the list of secret names for a given scope:

```
$ ssrun scope get path/to/my/secret
```

The next example assumes there is a file called **updatedMyTestSecretData1.txt** containing the data you want to use to update this secret.

3. Update a secret:

```
$ ssrun secret update -f updatedMyTestSecretData1.txt path/to/my/secrets:mytestsecret1
```

4. Retrieve a secret:

```
$ ssrun secret get path/to/my/secrets:mytestsecret1
```

5. Retrieve all secrets under a scope and save them in the directory "my_secret_dir"

```
$ ssrun secret get path/to/my/secrets -d my_secret_dir
```

6. Remove a secret:

```
$ ssrun secret delete path/to/my/secrets:mytestsecret1
```

> ***Note:*** *This not only removes the secret but also all metadata that is associated with it.*

7. Remove a scope:

```
$ ssrun secret delete path/to/my/secrets
```

> ***Note:*** *This not only removes the scope but also all scopes, secrets and metadata that are children of it.*

> For more information about how to log into DSSas root, please see "Initialize DevOps Secrets Safe" on page 5.

Manage Metadata

Before starting this section, ensure you have initialized, unsealed, and logged into DevOps Secrets Safe as root.

1. Create metadata for a secret:

```
$ ssrun metadata create -n mytestsecret1Meta1Name -v meta1Value
path/to/my/secrets:mytestsecret1
```

> ***Note:*** *When managing metadata, to reference a scope, set the URI to its path. For example,* ***path/of/scope****. To reference a secret, use* ***{path}:{secretName}****. For example,* ***path/of/scope:secretName****.*

2. Update metadata for a secret:

```
$ ssrun metadata update -n mytestsecret1Meta1Name -v updatedMeta1Value
path/to/my/secrets:mytestsecret1
```

3. View metadata for a secret:

```
$ ssrun metadata get -n mytestsecret1Meta1Name path/to/my/secrets:mytestsecret1
```

> ***Note:*** *The above command retrieves only the information associated with the metadata item named* ***mytestsecret1Meta1Name****. To retrieve the information for all metadata items associated with a scope or secret, omit the* ***-n*** *argument.*

4. Remove metadata:

```
$ ssrun metadata delete -n mytestsecret1Meta1Name path/to/my/secrets:mytestsecret1
```

> For more information about how to log into DSSas root, please see "Initialize DevOps Secrets Safe" on page 5.

Manage Safelists and IP Ranges

Before starting this section, ensure you have initialized, unsealed, and logged into DevOps Secrets Safe as root.

Safelists allow you to explicitly grant or deny access to specific IP addresses for all CLI commands. Safelists and IP ranges must be structured in the following way:

Safelist Model

- **Name**: (Required). Name for this safelist.
  - Names must be unique and can only include the following characters: 0-9, A-Z, a-z, underscore (_) and dash (-).
- **Description**: (Optional). Details about this safelist.

- **Expiry date**: (Optional). Specifies a day and time when this safelist will expire.
    - An empty or null value denotes no expiry.

IP Range Model

- **Name**: (Required). Name for this IP range.
    - Names must be unique to their parent safelist and can only include the following characters: 0-9, A-Z, a-z, underscore (_) and dash (-) .
- **Value**: (Required). Specifies a range of IP addresses.
    - The supported IP range value patterns are:
        - CIDR range: 192.168.0.0/24, fe80::%lo0/10
        - Single address: 10.101.8.16, fe80::1%23
        - Begin-end range: 10.101.8.10 - 10.101.8.20, fe80::1%23 - fe80::ff%23
- **Allow**: (Required). Specifies whether the defined range of IP addresses should be used to allow or deny access.
- **Description**: (Optional). Details about this IP range.
- **Expiry date**: (Optional). Specifies a day and time when this IP range will expire.
    - An empty or null value denotes no expiry.

***Note:** A safelist must have at least one IP range associated with it.*

1. Create two safelists:

```
$ ssrun safelist create -f safelist1.txt
```

```
$ ssrun safelist create -f safelist2.txt
```

The following examples assume there are two files, **safelist1.txt** and **safelist2.txt**, with the given contents:

Example:

safelist1.txt

```
{
        "ipRanges": [
           {
              "name": "ip_range_1",
              "value": "10.101.8.10-10.101.8.20",
              "allow": true,
              "description": "IP Range 1 Description",
              "expiryDate": "2020-06-21T11:44:31.733Z",
              "xForwardedForHeaderLimit": "2"
           }
        ],
        "name": "safelist_1",
        "description": "Safelist 1 Description",
        "expiryDate": "2020-06-21T11:44:31.733Z"
}
```

Note: *In the above example, the safelist is enforced only until the defined expiry date and allows only IP addresses in the range of 10.101.8.10 to 10.101.8.20.*

Example:

safelist2.txt

```
{
        "ipRanges": [
           {
              "name": "ip_range_2",
              "value": "10.101.8.50-10.101.8.60",
              "allow": false,
              "description": "IP Range 2 Description"
           }
        ],
        "name": "safelist_2",
        "description": "Safelist 2 Description"
}
```

Note: *In the above example, the safelist never expires and denies IP addresses in the range of 10.101.8.50 to 10.101.8.60.*

2. View safelists and IP ranges:

   The **safelist get** command will return all safelists that exist.

```
$ ssrun safelist get
```

You can also limit the view by passing in the name of the safelist targeted for discovery.

```
$ ssrun safelist get -n safelist_1
```

The **ip-range get** command will return all the ip ranges that exist for a given safelist.

```
$ ssrun ip-range get -n safelist_1
```

You can also limit the view by passing in the name of the IP range targeted for discovery.

```
$ ssrun ip-range get -n safelist_1 -i ip_range_1
```

Views can be further modified by using the following flags:

- **-d**: (Depth). Use this to define the maximum depth of the view to return.
  - A value of **0** returns only the element specified.
  - A value of **1** returns the element specified and all direct children.
  - A value of **2** returns all children and grandchildren of the element specified.
- **-v**: (Verbose). Use this to get a full listing of safelists and/or ip ranges attributes; otherwise, a slim view of each safelist or IP ranges is returned.

3. Update a safelist:

```
$ ssrun safelist update -n safelist_2 -f safelist2Update.txt
```

This command updates the safelist with the name **safelist_2**.

The following example assumes there is a file called **safelist2Update.txt** with the given contents:

Example:

safelist2Update.txt

```
{
        "description": "Safelist 2 Description Updated",
        "expiryDate": "2021-06-21T12:17:14.326Z"
}
```

4. Add an IP range to a safelist:

```
$ ssrun ip-range create -n safelist_2 -f ipRange.txt
```

This command adds an IP range to the safelist with the name **safelist_2**.

The following example assumes there is a file called **ipRange.txt** with the given contents:

Example:

ipRange.txt

```
{
        "value": "10.101.8.70",
        "allow": false,
        "description": "IP Range 3 Description",
        "expiryDate": "2021-06-21T11:58:03.315Z"
}
```

Note: *In the above example, the IP range is only enforced until the defined expiry date and denies IP requests coming from the IP address 10.101.8.70.*

5. Update an IP range of a safelist:

```
$ ssrun ip-range update -n safelist_2 -i ip_range_2 -f ipRangeUpdate.txt
```

This command updates the IP range with the name **ip_range_2** for the safelist with the name **safelist_2**.

The following example assumes there is a file called **ipRangeUpdate.txt** with the given contents:

Example:

ipRangeUpdate.txt

```
{
        "value": "10.101.8.71",
        "allow": false,
        "description": "IP Range 3 Updated",
        "expiryDate": "2021-06-21T11:58:03.315Z"
}
```

6. Assign a safelist to a user:

```
$ ssrun authorization create -p principal/internal/user/user1 -o read -a allow
safelist/safelist_2/access
```

This command associates the safelist with the name **safelist_2** to the user with the name **user1**.

7. Delete an IP range from a safelist:

```
$ ssrun ip-range delete -n safelist_2 -i ip_range_2
```

This command deletes the IP range with the name **ip_range_2** from the safelist with the name **safelist_2**.

8. Delete a safelist:

```
$ ssrun safelist delete -n safelist_2
```

This command deletes the safelist with the name **safelist_2**.

For more information about how to log into DSSas root, please see "Initialize DevOps Secrets Safe" on page 5.

Manage Event Sink Configurations

Create an event sink configuration:

```
ssrun event-sink create -f myconfig.json
```

This command creates an event sink configuration using the provided JSON file.

For detailed instructions on event sink configuration, please see "Event Sinks" on page 65.

Manage DevOps Secrets Safe CLI Contexts

Contexts are CLI-specific configurations that allow you to access multiple instance of DevOps Secrets Safe from a single client machine. CLI contexts exist only on the client side and only tell the CLI where to access the DevOps Secrets Safe instance. They do not interact with the instance in any way on their own.

Example:

*Assume you want to interact with two instances of DevOps Secrets Safe, one in staging and one in production, and you want to take the value of a secret from your staging DevOps Secrets Safe and save it to your production instance. In this example, your staging instance has an IP of **164.223.32.59** and your production instance has an IP of **164.225.37.62**.*

1. *Create a context pointing at staging:*

```
$ ssrun context create -n staging -a 164.223.32.59 -p 443 -s false -v v1
```

2. *Create a context pointing at production:*

```
$ ssrun context create -n production -a 164.225.37.62 -p 443 -s true -v v1
```

3. *Set the staging context to active:*

```
$ ssrun context set-current -n staging
```

4. *List all contexts:*

```
$ ssrun context get
```

```
CURRENT    NAME              HOSTNAME/IP      PORT  API VERSION    SSL CA
*          staging           164.223.32.59     443  v1             false
           production        164.225.37.62     443  v1             true
```

5. *Log in on the staging instance:*

```
$ ssrun login -u my_staging_user -p my_staging_user_password
```

6. *Save secret from staging DevOps Secrets Safe instance on your file system:*

```
$ ssrun secret get path/to/staging:secret -f mysecret
```

7. *Switch contexts so your CLI is pointing at the production DevOps Secrets Safe instance:*

```
$ ssrun context set-current -n production
```

8. *Log in with a user from the production DevOps Secrets Safe instance:*

```
$ ssrun login -u my_production_user -p my_production_user_password
```

9. *Create a new secret on the production instance storing the value retrieved from the staging instance:*

```
$ ssrun secret create path/to/production:secret -f mysecret
```

10. *Log in on the staging instance:*

```
$ ssrun login -u my_staging_user -p my_staging_user_password
```

11. *Save secret from staging DevOps Secrets Safe instance on your file system:*

```
$ ssrun secret get path/to/staging:secret -f mysecret
```

12. *Switch contexts so your CLI is pointing at the production DevOps Secrets Safe instance:*

```
$ ssrun context set-current -n production
```

13. *Log in with a user from the production DevOps Secrets Safe instance:*

```
$ ssrun login -u my_production_user -p my_production_user_password
```

14. *Create a new secret on the production instance storing the value retrieved from the staging instance:*

```
$ ssrun secret create path/to/production:secret -f mysecret
```

***Note:** The asterisk (*) in the **CURRENT** column of the staging entry shows it is the active context.*

***Tip:** To learn more about DevOps Secrets Safe CLI contexts you can use the **-h** flag (**$ ssrun context -h**).*

***Note:** Specific environment variables may override the current configured context.*

```
$ export SECRETSSAFE_HOST=(IP address or hostname of DevOps Secrets Safe instance)
$ export SECRETSSAFE_PORT=(port of DevOps Secrets Safe instance)
$ export SECRETSSAFE_VERIFY_CA=(bool indicating if ca should be verified)
```

IMPORTANT!

If any of the above environment variables are defined, they will override what is in the current context.

Install DevOps Secrets Safe

The DevOps Secrets Safe Kubernetes installation script performs several **kubectl** commands to insert data into the cluster and uses Helm v3 to install the application. In order for the application to run successfully, a cluster must exist and an Nginx Ingress Controller must be configured in the cluster. The installing user must provide BeyondTrust their DockerHub username in advance to be given permission to pull the required images.

Prerequisites

1. Kubernetes cluster with version 1.13, 1.14, 1.15, 1.16, or 1.17 available to host the deployment.
2. Install Kubectl and configure to allow full permissions to the cluster.
3. Install Helm and initialize with the appropriate Role-Based Access Control (RBAC).

For more information, please see the following:

- Install Kubectl on Linux
- Install Helm

Note: *As a reference deployment, DevOps Secrets Safe has been tested on a three-node Kubernetes cluster, each with a minimum of 6GB of RAM.*

Installation Instructions

The **install.sh** script can be run interactively or alternatively can be called with parameters to supply the required values. Any values not specified as parameters will be requested interactively.

To see a list of accepted parameters, run the install script with **--help**.

```
./install.sh --help
```

Note: *If an installation does not complete successfully, run the uninstaller before running the installer again.*

Example:

Install DevOps Secrets Safe using a postgreSQL database:

```
./install.sh --docker-hub-username docker-user --docker-hub-password dockerpass --docker-hub-email docker-user@beyondtrust.com --database-type postgres --connection-string 'Server=secretssafe.database.beyondtrust.com;Database=secrets-safe;Port=5432;User Id=postgresql-user@secretssafe;Password=postgresql-password;Ssl Mode=Require;'
```

Example:

Install DevOps Secrets Safe using an Oracle database:

```
./install.sh --docker-hub-username docker-user --docker-hub-password dockerpass --docker-hub-email docker-user@beyondtrust.com --database-type oracledb --connection-string 'User Id=oracleuser;Password=oraclepass;Data Source=10.10.10.10:1521/XE;'
```

Example:

Install DevOps Secrets Safe using a Microsoft SQL Server database:

```
./install.sh --docker-hub-username docker-user --docker-hub-password dockerpass --docker-hub-email docker-user@beyondtrust.com --database-type mssql --connection-string' Server=10.10.10.10;Database=secrets-safe;User Id=sqluser;Password=sqlpass;'
```

Upgrade Instructions

To upgrade DevOps Secrets Safe, first perform an uninstall followed by an installation using the install script from new deployment.

Uninstall Instructions

To remove a DevOps Secrets Safe installation from a cluster, run the uninstall script. The uninstall script will remove all DSS data, containers, secrets, etc. from the cluster. This does not include removing the database.

```
./uninstall.sh
```

Additional Notes - Nginx Ingress Installation

Currently the DevOps Secrets Safe application is compatible with the Nginx Ingress Controller.

If you wish to install this ingress controller from the official Helm chart for an on-premise deployment, the following command may be run:

```
helm install nginx-ingress stable/nginx-ingress --version v1.24.5 --namespace kube-system --set controller.hostNetwork=true --set rbac.create=true --set controller.kind=DaemonSet
```

If you wish to install this ingress controller from the official Helm chart for a cloud deployment, the following command may be run:

```
helm install nginx-ingress stable/nginx-ingress --version v1.24.5 --namespace kube-system --set controller.replicaCount=3 --set controller.service.externalTrafficPolicy=Local
```

Note: *The* ***--set controller.service.externalTrafficPolicy=Local*** *option is added to the Helm install command for safelist enforcement purposes. This will enable client source IP preservation for requests to containers in your cluster. If you do not plan to use safelist enforcement, this option can be exlcuded.*

Install the DevOps Secrets Safe CLI

The DevOps Secrets Safe CLI, **ssrun**, is a Python package that wraps functionality exposed by the DevOps Secrets Safe API into a convenient tool that is used to interact with the system.

Prerequisites

The DevOps Secrets Safe CLI should run on any major platforms supported by Python and which have Python 3.5 and pip3 or above available.

Install the Package with pip

The DevOps Secrets Safe CLI package, **secretssafe**, is installed and managed on a client machine by the Python package manager pip, through a **.whl** file supplied by Beyond Trust, and is located in the **CommandLineInterface** directory of the extracted archive.

Execute the following when running in a virtual environment:

```
$ pip install secretssafe-<version>-py3-none-any.whl
```

Conversely, execute the following when running outside a virtual environment:

```
$ pip3 install secretssafe-<version>-py3-none-any.whl
```

Execute the CLI

After a successful installation, the CLI may be run by executing the following from any location on the filesystem:

```
$ ssrun
```

***Note:** If the **secretssafe** package is installed inside a virtual environment, the environment must be first activated for **ssrun** to be on the path and thus executable.*

Configure the Initial Context

Contexts allow for multiple DevOps Secrets Safe instances to be easily configured and accessed from a single client machine. On preliminary installation, execute the following to be prompted for details of the initial context:

```
$ ssrun context create
```

Follow the prompts to configure the DevOps Secrets Safe instance that the CLI will initially interact with. To view your configured clusters, execute the following:

```
$ ssrun context get
CURRENT    NAME       HOSTNAME/IP      PORT  API VERSION    SSL CA
*          localhost  localhost        8443  v1             false
```

The initial context will be set to **current** (configuration to use during any other CLI action) on creation, and any subsequent contexts created may be configured as **current** with the following command:

```
$ ssrun context set-current -n <context_name>
```

In addition, specific environment variables may be used to override the current context:

```
$ export SECRETSSAFE_HOST=<IP address or hostname of Secrets Safe instance>
$ export SECRETSSAFE_PORT=<port of Secrets Safe instance>
```

Note: *The following variable is necessary only if the certificate authority is not publicly trusted.*

```
$ export SECRETSSAFE_VERIFY_CA=<path_to_ca_cert>
```

Note: *The DevOps Secrets Safe CLI verifies the SSL certificate presented by the DSS instance. The* ***SECRETSSAFE_VERIFY_CA*** *environment variable or* ***SSL CA*** *context attribute specifies the path to the CA certificate that the DSS certificate is checked against.*

If no **SECRETSSAFE_VERIFY_CA** is specified, the default certificate bundles provided by the Python requests library are used.

Certificate verification can be disabled by setting **SECRETSSAFE_VERIFY_CA=false**. This is strongly discouraged for production environments.

To use these environment variables by default, rather than manually managing contexts, you can make them persistent in the shell environment. They can be stored in a users **~/.bashrc** file.

Example:

```
$ echo 'export SECRETSSAFE_HOST=1.1.1.1' >> ~/.bashrc
$ echo 'export SECRETSSAFE_PORT=443' >> ~/.bashrc
$ echo 'export SECRETSSAFE_VERIFY_CA=false' >> ~/.bashrc
$ source ~/.bashrc
```

Note: *In the example above, certificate verification has been set to* ***false****. While this is convenient for testing, it is not recommended in a production environment.*

Bash Autocompletion

The DevOps Secrets Safe CLI comes with the ability to configure bash autocompletion for ease of use and convenience. To install bash completion globally, execute the following:

```
$ ssrun completion bash > /etc/bash_completion.d/ssrun
```

This will allow any new bash instances to autocomplete the DevOps Secrets Safe CLI commands on demand. Sudo rights may be required to be able to write to **/etc/bash_completion.d/**.

API View Model

View Model Options

The DevOps Secrets Safe View Model consists of the following options:

- **Verbosity**: When used, returns a full attribute listing of whatever element is being discovered. Otherwise, a slim view of the element is returned.
- **Depth**: The maximum depth of the view to return. A value of **0** returns only the element being discovered. A value of **1** returns the element specified and all direct children. A value of **2** returns all children and grandchildren of the element being discovered.
- **Pagination**: Pagination consists of two options:
    - **Page Size**: The number of records to return for the direct children of the element being discovered. Values must be between 1 and 100.

> ***Note:*** *All grandchildren of the element being discovered will also be limited to this page size if it is specified, otherwise, it will be limited to the maximum page size of 100.*

    - **Page Number**: The page number (1-based) of results to return.
- **Sorting**: Specifies the sort order to use with the elements being discovered.

View Model Usage

Each sub command **get** operation supports all or some subset of the following view model options:

- **-v, --verbose**: Verbosity is optional and defaults to **false**.
- **-d, --depth**: Depth is optional and defaults to **1**.
- **-ps, --page-size**: Page size is optional and defaults to **50**. Invalid values are forced within the range of **1** and **100** (maximum page size).
- **-pn, --page_number**: Page number is optional and defaults to **1**. Invalid values are forced within the range of **1** and the total page count.
- **-sb, --sort-by**: Sorting is optional, and by default, no sorting is applied. This is defined as a single field or comma-delimited list of fields with the sort order (one of: **asc** or **desc**) specified in brackets after each.

> ***Example:***
>
> ***Name(desc)*** *OR* ***Name(desc),Url(asc)***

To determine what options are supported for each sub command **get** operation, use the **-h (help)** flag.

Example:

```
$ssrun user get -h
```

```
Usage: ssrun user get [-h] [-n USERNAME] [-i IDENTITY_PROVIDER] [-v] [-ps PAGE_SIZE] [-pn
PAGE_NUMBER] [-sb SORT_BY]

optional arguments:
       -h, --help              : Show this help message and exit.
       -n, --name              : If a name is specified, only that user is returned and
                                 sorting and pagination options are ignored. If no name is
                                 specified, a list of users is returned with sorting and
                                 pagination options (if specified) applied.
       -i, --identity-provider : Identity provider name. Defaults to internal if not
                                 specified.
       -v, --verbose           : Verbose output. Use the -v flag to get a full listing of
                                 users attributes. Otherwise, a slim view of each user is
                                 returned.
       -ps, --page-size        : Specifies the maximum number of elements to return in the
                                 user listing. Value must be between 1 and 100. All
                                 membership or group listings are also limited to this page
                                 size.
       -pn, --page_number      : Specifies the page number (1-based) of results to return.
       -sb, --sort-by          : Specifies the sort order for the users list. This is
                                 defined as a single field or comma delimeted list of fields
                                 with the sort order (one of:ace|desc) specified in brackets
                                 after each. For example: Name(desc) OR Name(desc),Url(asc).
```

Note: *In the above example, the* ***get*** *operation for the* ***user*** *sub command does not support the depth option. Whereas, it does for the* ***safelist*** *sub command:*

Example:

```
$ ssrun safelist get -h
```

```
optional arguments:
       -h, --help            : Show this help message and exit.
       -n, --name            : If a name is specified, only that safelist is returned and
                               sorting and pagination options are ignored. If no name is
                               specified, a list of safelists is returned with sorting and
                               pagination options (if specified) applied.
       -v, --verbose         : Verbose output. Use the -v flag to get a full listing of
                               safelist and IP range attributes. Otherwise, a slim view of
                               each safelist and ip range is returned.
       -d, --depth           : The maximum depth of the view to return. A value of 0 returns
                               only the element specified. A value of 1 returns the element
                               specified and all direct children. A value of 2 returns all
                               children and grandchildren of the element specified.
       -ps, --page-size      : Specifies the maximum number of elements to return in the
                               safelist listing. Value must be between 1 and 100. All IP range
                               listings are also limited to this page size.
       -pn, --page_number    : Specifies the page number (1-based) of results to return.
       -sb, --sort-by        : Specifies the sort order for the safelist list. This is
                               defined as a single field or comma delimeted list of fields with
                               the sort order (one of:ace|desc) specified in brackets after each.
                               For example: Name(desc) OR Name(desc),Url(asc).
```

Discovery Results

When paging is applied to a result set, the output always contains the following block of pagination information:

```
"Paging": {
       "CurrentPage": 1,
       "PageCount": 15,
       "PageSize": 10,
       "TotalSize": 147
}
```

- **CurrentPage**: The current page of the results that were discovered.
- **PageCount**: The page number of the results that were returned.
- **PageSize**: The total the number of results that were returned.
- **TotalSize**: The total number of elements that exist.

In the example below, we specify that we want to use a page size of ten, that we want to view the second page of that result set, and that we want to sort the results in ascending order by name:

Example:

```
$ ssrun user get -ps 10 -pn 2 -sb 'Name(asc)'
```


```
{
        "Uri": "/principal/internal/user",
        "Users": [
                {
                        "Uri": "/principal/internal/user/Berta"
                },
                {
                        "Uri": "/principal/internal/user/Bob"
                },
                {
                        "Uri": "/principal/internal/user/Boomer"
                },
                {
                        "Uri": "/principal/internal/user/Brad"
                },
                {
                        "Uri": "/principal/internal/user/Brenda"
                },
                {
                        "Uri": "/principal/internal/user/Catherine"
                },
                {
                        "Uri": "/principal/internal/user/Cathy"
                },
                {
                        "Uri": "/principal/internal/user/Chloe"
                },
                {
                        "Uri": "/principal/internal/user/Chris"
                },
                {
                        "Uri": "/principal/internal/user/Cicilia"
                }
        ],
        "Paging": {
                "CurrentPage": 2,
                "PageCount": 15,
                "PageSize": 10,
                "TotalSize": 147
        }
}
```


Note: *In the above example, the paging information informs us that there are a total of 15 pages available for discovery (given the page size we specified), and there are a total of 147 users.*

Access Control

Before starting this section, ensure a new instance of DevOps Secrets Safe is running and the DSS CLI is configured to communicate with it. In addition, the system should be initialized, unsealed and a successful root authentication executed.

> For more information about how to install the DevOps Secrets Safe CLI, please see Install the DevOps Secrets Safe CLI.
>
> For more information about how the system should be initialized, unsealed, and successful root authentication, please see Getting Started with DevOps Secrets Safe.

The creation of users and applications introduces to the system the concept of principals (authorizable entities) that are subject to access control with relation to other resources in the system. This may be in the form of denying or granting a principal access to a given secret, change their own password, or the ability to create other principals.

This can be achieved as described in the Getting Started with DevOps Secrets Safe guide, or by utilizing groups.

Groups, as supported by DevOps Secrets Safe, provide the ability to grant or deny access to a resource by association. This reduces the overhead of multiple API calls to achieve bulk access control for multiple users of the system.

Create Users

Create a user with the following command:

```
$ ssrun user create
```

When prompted, provide the name to identify the user. This will return the principal details of the user, along with the URI.

Create Groups

Create a group with the following command:

```
$ ssrun group create
```

When prompted, provide a name to identify the group. This will return the principal details of the group, along with the URI.

> ***Note:*** *Although a group is itself identified as a principal, groups may NOT be added to groups.*

> For more information about how to configure DevOps Secrets Safegroups with membership managed by external identity providers, please see Identity Provider Configuration.

Add or Remove Principals

Add principals by supplying a comma-separated list of principal resources when prompted:

```
$ ssrun group add-member
```

Optionally, you may enter the group name and principal resource list with keyword arguments:

```
$ ssrun group add-member --name <group_name> --principal_resources <principal_resources_list>
```

Similarily, to remove principals from a group, supply a comma-separated list of principal resources while prompted to the following command:

```
$ ssrun group remove-member
```

You may also execute the command promptless as follows:

```
$ ssrun group remove-member --name <group_name> --principal_resources <principal_resources_list>
```

You may supply principal resources for both applications and users in a single command. A successful call to these commands will result in no output from the CLI, while a failure will output the error.

Delete Groups

Delete a group with the following command:

```
$ ssrun group delete
```

When prompted, provide the principal name to identify the group. This will return the principal details of the group, along with the URI.

Note: *The deletion of a group will also remove all authorization associations provided by the creation of the grouping.*

Query Group Membership

The principal discovery mechanism, as described in Getting Started with DevOps Secrets Safe, allows for the querying of both the groups a particular principal belongs to and the principals belonging to a particular group.

List Members of a Group

To list the members of a particular group, execute the following command:

```
$ ssrun group get --name <group_name> --verbose
```

Sample Output:

```
{
    "Uri": "/principal/internal/group/root",
    "ID": 2,
    "Name": "root",
    "Type": "group",
    "RemoteId": "c0d7cc51-83fb-4aa1-98fa-0a5b398f0132",
    "IdentityProvider": "internal",
    "IsRoot": true,
```

```
    "GroupMembers": [
        {
            "Uri": "/principal/internal/group/root/members",
            "Name": "members",
            "Members": [
                {
                    "Uri": "/principal/internal/user/root",
                    "ID": 1,
                    "Name": "root",
                    "Type": "user",
                    "RemoteId": "49d98c1d-29d4-450a-b14a-167cdb63b233",
                    "IdentityProvider": "internal",
                    "IsRoot": true
                }
            ]
        }
    ]
}
```


Note: *the creator of a group is given read access by default to the members resource. This does not need to be explicitly granted after group creation, except to other querying principals.*

List Groups a Principal Belongs To

To list the groups a particular principal belongs to, execute the following command:

```
$ ssrun <user/application> get --name <principal_name> --verbose
```

Sample Output:


```
{
    "Uri": "/principal/internal/user/root",
    "ID": 1,
    "Name": "root",
    "Type": "user",
    "RemoteId": "49d98c1d-29d4-450a-b14a-167cdb63b233",
    "IdentityProvider": "internal",
    "IsRoot": true,
    "UserGroups": [
        {
            "Uri": "/principal/internal/user/root/groups",
            "Name": "groups",
            "Groups": [
                {
                    "Uri": "/principal/internal/group/root",
                    "ID": 2,
                    "Name": "root",
                    "Type": "group",
                    "RemoteId": "c0d7cc51-83fb-4aa1-98fa-0a5b398f0132",
                    "IdentityProvider": "internal",
```

```
                    "IsRoot": true
                }
            ]
        }
    ]
}
```

Note: *As with listing members, a user is given read access by default to its own groups resource.*

Manage Access Control by Group Association

In all authorization checks, an authenticated user or application will provide a list of associated principals in which to determine access to an operation on a particular resource. Principals inherit all access permissions configured for groups they are members of. Denial-type rules take precedence over allow-type rules whenever access control entries conflict. Users and applications may exist in multiple groups, allowing for building comprehensive access control rules.

To configure authorization for a particular group (**assume group URI principal/internal/group/<group_name>**), execute an authorization creation command for a particular resource (**assume secret URI secret/testsecret:mytestsecret**), as described in Getting Started with DevOps Secrets Safe.

Allow Resource Access by Group

```
$ ssrun authorization create secret/testsecret:mytestsecret -p principal/internal/group/<group_name>
-o read -a allow
```

If a given application or user belongs to group with group name <group_name>, they will inherit the read permission on the secret at URI **secret/testsecret:mytestsecret**.

Deny Resource Access by Group

Conversely, a group may be given an explicit denial to a resource, causing all group members to inherit the denial of access:

```
$ ssrun authorization create secret/testsecret:mytestsecret -p principal/internal/group/<group_name>
-o read -a deny
```

This will override non existing access, along with explicit allow access. This allows the administrator to build user groups which can be shielded from manipulating or reading sensitive existing resources in DevOps Secrets Safe.

Configuration Settings

Configuration settings can be modified at runtime by using the CLI or by using API calls. Configuration settings are applied globally, meaning they apply for all users of DevOps Secrets Safe.

List All Configuration Settings

```
ssrun setting get
```

Outputs a list of configured settings in JSON format.

Example:

```
{
        "Server:Limits:MaxRequestBodySize": 10000,
        "Jwt:AccessTokenTimeToLiveSeconds": 3600,
        "Jwt:RefreshTokenTimeToLiveSeconds": 2592000,
        "JwtConfigs:Local:ClockSkewSeconds": "300"
}
```

List One Configuration Setting

```
ssrun setting get -n <setting-name>
```

Outputs the value of an individual setting.

Example:

```
Server:Limits:MaxRequestBodySize = 10000
```

Delete a Configuration Setting

```
ssrun setting delete -n <setting-name>
```

Deletes the setting with the specified name.

Create a Configuration Setting

```
ssrun setting create -n <setting-name> -v <setting-value>
```

Creates a configuration setting using the values specified on the command line.

Update a Configuration Setting

```
ssrun setting update -n <setting-name> -v <setting-value>
```

Updates a configuration setting that already exists, using the values specified on the command line.

Delegate Permissions for Configuration Settings

```
ssrun authorization create -p principal/internal/user/<User> -o create -a allow system/settings/
```

Grants permission to create configuration settings for a user that does not have root user permissions. Permissions to read, update, and delete can also be granted.

Current Configuration Settings

Server:Limits:MaxRequestBodySize

The maximum body size (in bytes) for a request. This setting is applied to the gateway service and will reject requests with larger than configured message body size. The default is 1048576 bytes.

Jwt:AccessTokenTimeToLiveSeconds

Used to set the access token time to live (in seconds). The minimum value is 10 seconds. The default is 3600 seconds.

Jwt:RefreshTokenTimeToLiveSeconds

Used to set the refresh token time to live (in seconds). The minimum value is 0 seconds. The default is 2592000 seconds.

JwtConfigs:Local:ClockSkewSeconds

Used to set the access token time to live skew (in seconds). When the access token is validated this setting allows compensating for server time drift. The default is 300 seconds.

> ***Note:*** *Changing this setting could result in unauthorized 401 responses if there is a time difference between the token time and the DevOps Secrets Safe services time.*

Jwt:TokenCleanupIntervalSeconds

Used to control the frequency of cleaning up old Jwt tokens. The default is 3600 seconds.

> ***Note:*** *Changing this setting could have performance repercussions.*

DevOps Secrets Safe Integrations

BeyondTrust DevOps Secrets Safe offers integrations with the following third-party products:

- Ansible Integration
- Azure Devops Integration
- Jenkins Integration
- Kubernetes Integration
- Puppet Integration

Ansible Integration

Before proceeding with this section, please ensure that you have access to the **secretssafe** Python package, installable from a BeyondTrust provided **.whl** file.

Install the DevOps Secrets Safepackage

The DevOps Secrets Safe lookup plugin imports the **secretssafe** package and creates an instance of the client which communicates with the DevOps Secrets Safe cluster.

Install the **secretssafe** package to your Python environment using pip.

```
$ pip install secretssafe-<version_details>.whl
```

Configure Ansible to Discover the DevOps Secrets Safe Lookup Plugin

To use the DevOps Secrets Safe lookup plugin, you will need to either export a particular environment variable to point to the location of the plugin *.py file, or place the plugin ***.py** file in one of the ansible "magic" directories.

To load the plugin automatically, store it in **~/.ansible/plugins/lookup**, **/usr/share/ansible/plugins/lookup**, or place the path to the plugin in your **ansible.cfg** file.

To use the plugin only in certain playbooks, store it in sub directory named **lookup_plugins** in the directory that contains the playbook that utilizes the plugin.

To use the environment to configure the plugin location, export the following:

```
$ export ANSIBLE_LOOKUP_PLUGINS=<path/to/secretssafe/lookup/plugin/directory/>
```

Once properly configured, validate the discovery of the plugin:

```
$ ansible-doc -t lookup secretssafelookup
```

The lookup plugin will then be invocable within a playbook similar to any other lookup plugin that come with the default Ansible installation.

Execute the Plugin with Environment Variables

The plugin allows for the usage of environment variables for the configuration of the client and authentication of the calling process, along with the keyword arguments as described in the plugin documentation. The following variables will be need to be set either on the control machine (shell where ansible is called), or within the playbook that uses the plugin:

```
SECRETSSAFE_HOST=<IP address or hostname of Secrets Safe instance>
SECRETSSAFE_PORT=<port of Secrets Safe instance>SECRETSSAFE_API_KEY=<pregenerated API key>
SECRETSSAFE_APP_NAME=<application name associated with API key>
SECRETSSAFE_VERIFY_CA=<true/false/path to CA certificate>
```

This allows you to invoke the plugin without the credential/configuration keyword arguments.

Note: *The DevOps Secrets Safe client verifies the SSL certificate presented by the DSS instance. The* ***SECRETSSAFE_VERIFY_CA*** *environment variable specifies the path to the CA certificate that the Secrets Safe certificate is checked against.*

If no ***SECRETSSAFE_VERIFY_CA*** *is specified, the default certificate bundles provided by the Python requests library are used.*

Certificate verification can be disabled by setting ***SECRETSSAFE_VERIFY_CA=false****. This is strongly discouraged for production environments.*

DevOps Secrets Safe create_secret Module

This module supports creating secrets with DSS. Secrets can be read from a file on disk or an Ansible fact. Secret creation by providing a generator name is also supported. Credentials for a DSS application must be provided.

Options

- **name:** The name of the application used to create the secret
  - required: true
- **api_key:** API key that corresponds to the application provided in the name option
  - required: true
- **secret_uri:** The DSS URI where the secret will be saved
  - required: true
- **host:** DNS or IP address of the DSS instance this secret will be saved to
  - required: true
- **verify_ca:** SSL certificate verification flag; looks to publicly available CA if set to **true**
  - required: false
  - default: **true**
  - choices:
    - **true**
    - **false**
    - **path to CA certificate**
- **port:** DSS instance port
  - required: false
  - default: 443
- **generator:** Name of generator used to create secret value mutually exclusive with **secret_file_path** and **secret_value** options
  - required: false
- **secret_file_path:** Path to file that will be saved as a secret mutually exclusive with **generator** and **secret_value** options
  - required: false

- **secret_value:** Value, in the form of a string, that will be saved as a secret path to file that will be saved as a secret mutually exclusive with **generator** and **secret_file_path** options
    - required: false

Azure DevOps Integration

Retrieve Secrets from DevOps Secrets Safe

This extension allows for the retrieval of ASCII secrets from an Azure-accessible instance of DevOps Secrets Safe.

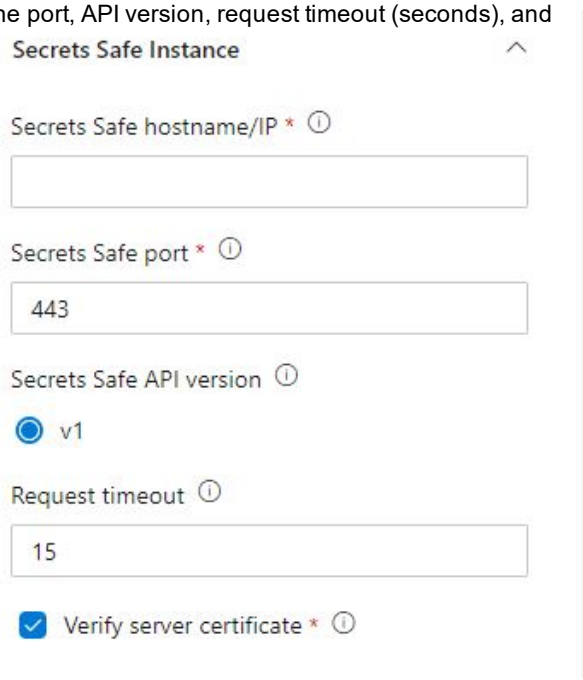
Prerequisites

In order for this extension to retrieve a secret for use in a given Azure DevOps pipeline, the DevOps Secrets Safe instance must be preconfigured with the secret in question and an application principal authorized to read it. The URI of the secret and both the application name and API key assigned to the application are required as input values for this extension.

Secrets Safe Instance Configuration

Enter the public hostname/IP of the DevOps Secrets Safe instance, along with the port, API version, request timeout (seconds), and server certificate verification flag. The default values are shown in the provided image.

> ***Note:*** *The build agents require access to the certificate authority used to sign the certificate used by the DevOps Secrets Safe cluster ingress service, be it a publicly available certificate or installed to the build agent itself.*



Authentication

Enter the name of the application authorized to read the requested secrets, along with the associated API key. The default application name is **azure-devops**, unless specified.

Secret

Enter the URI of the requested secret, and the name of the pipeline variable to populate. If this variable is configured as secret, then this extension both populates the value and retains the secret state, not logging the output to the task log. The secret variable can then be used in a subsequent task in the pipeline without ever exposing the value.

> ***Note:*** *Multi-line values are allowed only if the storage variable is not marked secret. Azure DevOps secret pipeline variables only support single line secrets, and the DevOps Secrets Safe secret retrieval will fail if a secret is multilined and requested to populate a secret pipeline variable.*

Jenkins Integration

Installation

The plugin is packaged as a self-contained **.hpi** file which can be installed either from the web UI or via the Jenkins CLI. Once you have acquired the file, **devops-secrets-safe.hpi**, proceed with one of the following installation methods.

> For more information, please see https://jenkins.io/doc/book/managing/plugins/.

Via Jenkins Web UI

The most common method for plugin installation and administration is to use the web UI.

1. Authenticate as a user with administrative permissions and navigate to **Manage Jenkins > Manage Plugins**.
2. Once there, click the **Advanced** tab and scroll down to the **Upload Plugin** section.
3. Click **Choose File** to browse to and select your **.hpi** file.
4. Finally, click **Upload** and allow Jenkins to restart once installation has finished.

Via the Jenkins CLI

The Jenkins CLI can also be leveraged for many administrative tasks including plugin installation. To install via the CLI, execute a command similar to:

```
java -jar jenkins-cli.jar -s "http://your-jenkins-server:8080/" install-plugin "path/to/devops-secrets-safe.hpi" -deploy -restart
```

Configuration

It is important to note that the plugin can be configured at any or all of the available scopes within a Jenkins environment. This means that configuration can exist at the global level (**Manage Jenkins > Configure System**), at the folder level, or at the individual item or project level.

When a build job executes, configuration is resolved starting at the most specific scope and working back up the chain until a valid (fully-populated) configuration is found: **item level > folder level (through multiple folder levels if present) > global level**.

The specific information collected for configuration includes the following fields:

- **Name / Alias**: Provides a place to give a quick, descriptive name to the collection of settings.
- **DSS URL**: The base URL of the DSSinstance including protocol, hostname or IP, and port.
- **DSS Application Credentials**: The application credentials used to authenticate to the DSS RESTful APIs.
- **Skip SSL Validation**: If enabled, the plugin will skip validation of any SSL cert presented by the DSS instance during the execution of RESTful API calls.

The credentials used for authentication to the DSS instance are stored in the Jenkins internal credential store and read at the time of job execution. They are stored as a custom credential type named **DSS Application Credentials** which require an **Application Name** and **API Key** matching a configured principal within DSS.

Usage

Secrets are retrieved from DevOps Secrets Safe for use in a build based on information provided in each project's build configuration, injected as environment variables, and intentionally limited in scope to help avoid exposing them outside of where they are actually used.

The following is an example of the configuration necessary to retrieve and use secrets within a build process:

```
def requestedSecrets = [
       [ secretUri: 'full/scope/path:git-user', environmentVariable: 'gitPwd' ],
       [ secretUri: 'full/scope/path:admin-user', environmentVariable: 'adminPwd' ]
];
withDss(requestedSecrets: requestedSecrets) {
       // ..... do some build stuff
       bat my_program.exe -u git-user -p ${env.gitPwd}
       // ..... more build stuff
       bat my_other_program.exe --administratorPassword "${env.adminPwd}"
}
```

In the above example, the **withDss** block defines the scope within which the secrets will be available and initiates the retrieval of those secrets. The required parameter for **withDss() {...}** is **requestedSecrets**, which should be supplied with an array of secrets you wish to retrieve and use within the block.

The individual entries in the **requestedSecrets** array should contain two properties:

- **secretUri**: The full path / scope for the secret followed by a colon and the secret name.
- **environmentVariable**: The environment variable name by which you'd like to reference the secret value within the block.

To access the secret values, simply reference them as you would any other environment variable in your script:

```
${env.variable-name}
```

Note: *The values are also accessible via the secret URI as follows:*

```
${env.'full/scope/path:secret-name'}
```

Again, the values are only available within the **withDss** block and will be retrieved from DevOps Secrets Safe using the most specific configuration that can be resolved by the plugin for the given job.

Kubernetes Integration

The Kubernetes integration for DevOps Secrets Safe (DSS) enables injection of secrets from DSS into Kubernetes pods.

Injecting DSS Secrets Into Kubernetes Pods via a Sidecar

You can configure pods to retrieve secrets from DSS before starting their primary application. The application consuming the secrets does not need to have any knowledge of DSS to use the secret at runtime.

This integration uses the Kubernetes *service account* construct as the basis for identity in DSS. Permissions for access to secrets in DSS can be granted to specific service accounts that are represented by user principal identities within DSS.

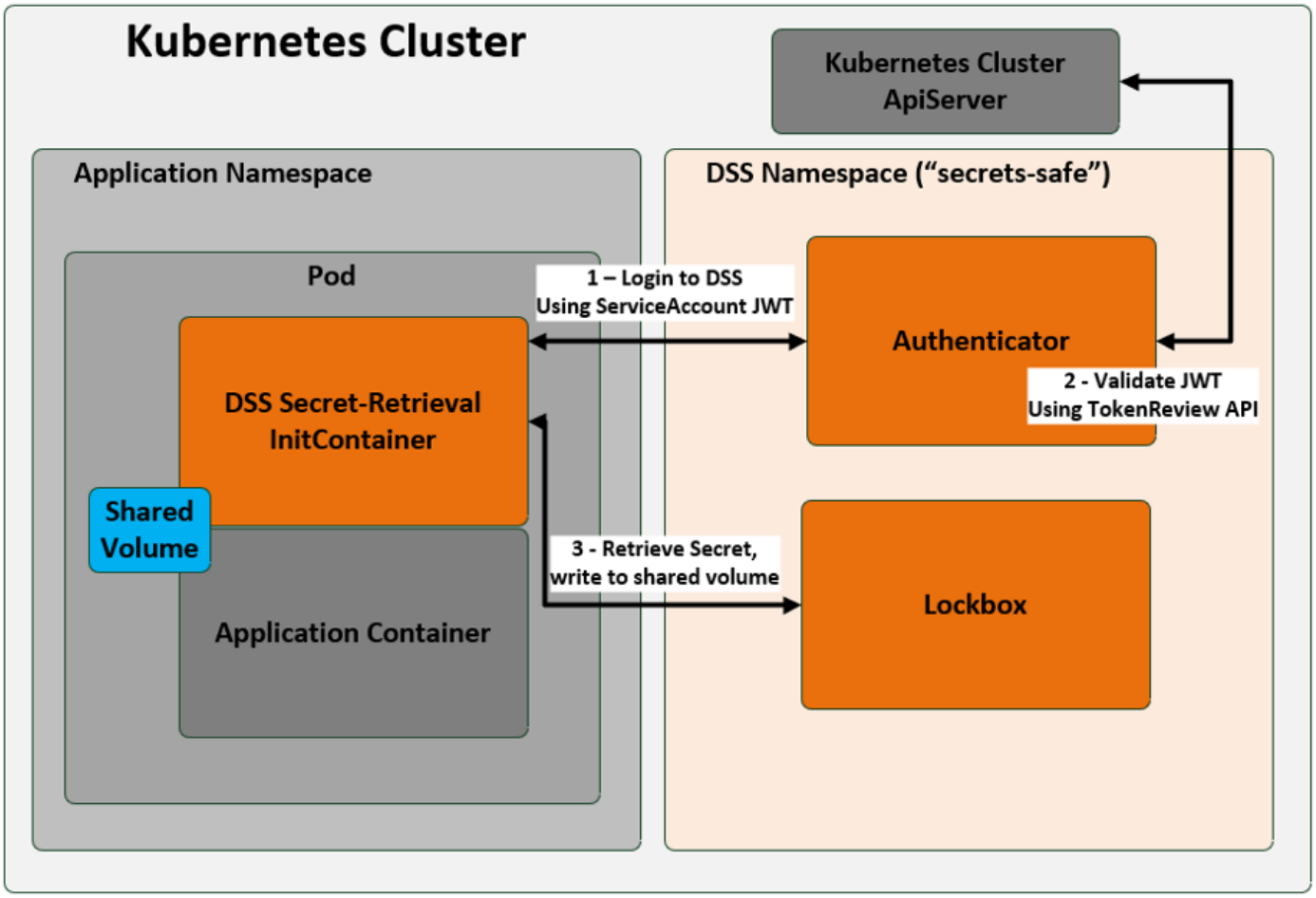
Overview

Applications can opt in to DSS secret retrieval by adding the DSS **secrets-agent** to their Kubernetes manifests as an init container or a sidecar. The **secrets-agent** retrieves secrets and makes them available to the target consumer without requiring that consumer to be aware of DSS.

To support this feature, the DSS instance must have a Kubernetes-type identity provider configured, and the Kubernetes **ServiceAccount** assigned to the DSS application pods must be granted access to the **TokenReview** API for the cluster.

At runtime, secret retrieval from DSS by Kubernetes pods follows the order of operations pictured in the figure below:

1. The **InitContainer** requests a DSS authentication token using the **ServiceAccount** JWT for the pod it is running inside of.
2. DSS validates the **ServiceAccount** JWT using the **TokenReview** API on the cluster's API server. In order to authenticate to the cluster's API server, DSS assumes the identity of its own pod's **ServiceAccount** when communicating with the Kubernetes API. The DSS **ServiceAccount** requires additional role-based access control (RBAC) permissions on the cluster for it to access the cluster's **TokenReview** API.
3. The **secret-retrieval** container requests the secret contents from DSS and writes them to a volume that is shared with the application container. In this case, and **init-container** pattern is being used; **secrets-agent** exits, and the application container begins execution with its target secret contents from DSS available on its file system at the shared volume path.



This guide provides a walkthrough of the steps required to configure and demonstrate this integration. The walkthrough regularly assumes the following default installation parameters for the DSS instance and cluster being used:

- DSS is installed in the **secrets-safe** namespace of the Kubernetes cluster.
- The TLS certificate presented by the DSS instance is trusted (not self-signed).
- The Kubernetes identity provider for DSS is configured with the default name **kubernetes**.

> ***Note:*** *Devaitions from any of these assumptions necessitate extra configuration described in the "Detailed Manifest Example" on page 48.*

If DSS is not installed into the **secrets-safe** namepsace, then replace the namespace identifier string **secrets-safe** in configurations and account names with the alternative namespace used for DSS.

DSS Configuration - Kubernetes Identity Provider

The Kubernetes identity provider enables authentication to DSS using Kubernetes **ServiceAccount** tokens. Kubernetes **ServiceAccounts** become user principals in DSSthat use the **ServiceAccount** JWT as their login credential.

The Kubernetes identity provider requires only minimal configuration within the DSS instance for this in-cluster use case, because DSS services are running inside pods and can obtain the cluster's parameters from their pod's environment.

Example: *A sample configuration for the Kubernetes identity provider in DSS for this use case is the following:*

```
{
        "Name": "Kubernetes",
        "Type": "Kubernetes",
        "Enabled": true
}
```

Note: *Additional details and configuration use cases for this identity provider are described in the "Identity Provider Configuration" on page 52 document.*

Cluster Configuration - Add RBAC Permissions for the DSS ServiceAccount

In order for DSS to validate **ServiceAccount** tokens, DSS itself assumes the identity of its pod's **ServiceAccount**. This DSS **ServiceAccount** requires additional RBAC permissions on the cluster for it to access the cluster's **TokenReview** API and to list the cluster's ServiceAccounts.

This example assumes that DSS is installed in the namespace **secrets-safe** and that the DSS pods are using the **default ServiceAccount** within that namespace. If DSS is installed in a different namespace, replace **secret-safe** in the examples with the namespace used. If a service account different from **default** is assigned to DSS, use that service account's name instead.

The RBAC permissions required for DSS **secret-retrieval** integration are defined in the following manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dss-kubernetes-identity
rules:
- apiGroups: [""]
  resources: ["serviceaccounts"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["authentication.k8s.io"]
  resources: ["tokenreviews"]
  verbs: ["create"]
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: dss-kubernetes-identity-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: dss-kubernetes-identity
subjects:
  - kind: ServiceAccount
```

```
      name: default
      namespace: secrets-safe
```

After saving the above snippet to a **.yaml** file named **secret-retrieval-privileges.yaml**, apply it to the target cluster using the command:

```
kubectl apply -f secret-retrieval-privileges.yaml
```

> ***Note:*** *The Kubernetes API Server must be running with* ***--service-account-lookup*** *to ensure that deleted* ***ServiceAccount*** *tokens are properly revoked. If this setting is not enabled, JWTs for deleted* ***ServiceAccounts*** *remain as valid credentials. This setting is enabled by default as of Kubernetes 1.7.*

Configuring an Application for Secret Retrieval

In order for an application to opt in to DSS secret retrieval:

- DSS authorization for the application pod's identity must be configured.
- The application's manifest must have the DSS **secrets-agent** container and **Shared Volume** added.

This section first presents a guide to setting up the required authorization in DSS to allow a pod's **ServiceAccount** to retrieve secrets, then shows the pod manifest modifications necessary to add the **secret-retrieval** behavior to the pod at runtime.

DSS Permissions - Authorize Access to the Target Secret

The **ServiceAccount** for the application must be authorized to access its target secrets in DSS. This is accomplished by the following steps:

1. Create the ServiceAccount principal in DSS.

   There are two means for creating a principal in DSS for the ServiceAccount: pre-creation using the ServiceAccount name, or creation at first login to DSS using the ServiceAccount's JWT.

   Use the ServiceAccount Name

   DSS principals for ServiceAccountrs can be created using the ServiceAccount name.

   The ServiceAccount name is formatted according to Kubernetes RBAC subject rules, using **.** as the separator instead of **:**.

   Format the serviceaccount for DSS as:

   ```
   system.serviceaccount.<namespace>.<account-name>
   ```

   To print a list of all ServiceAccounts for a cluster in this format, at the bash console for the Kubernetes cluster administrator, run the command:

   ```
   OLD_IFS=$IFS; IFS=$'\n'; for i in `kubectl get serviceaccount --all-namespaces`; do echo -n
   "system.serviceaccount."; echo -n $i | awk '{printf $1}'; echo -n "."; echo -n $i | awk '{print
   $2}'; done; IFS=$OLD_IFS
   ```

The list of serviceaccounts is returned in the format:

```
system.serviceaccount.NAMESPACE.NAME
```

Knowing the name of the target service account, the DSS principal can be created using the **ssrun user create** CLI command.

For example, to create a DSS principal for the default ServiceAccount in the **secrets-safe** namespace, use the command:

```
ssrun user create -i kubernetes -n system.serviceaccount.secrets-safe.default
```

Use ServiceAccount JWT

Logging in to DSS using the ServiceAccount JWT requires obtaining the ServiceAccount JWT from the cluster.

The JWT for the ServiceAccount can be obtained from the cluster using kubectl as follows:

```
$ kubectl get secret <name-of-token-secret> -n <application-namespace> -o jsonpath="
{.data.token}"| base64 --decode
```

In the above command, **<name-of-token-secret>** is the name of the Kubernetes secret for the target **ServiceAccount** credential, and **<application-namespace>** is the cluster namespace where the target application exists. If the **secret-retrieval** application is installed in the default namespace, the **-n** parameter can be omitted.

Use the obtained JWT as the **password** credential at the DSS connect/token endpoint, targeting the external identity provider named **Kubernetes**, if following this example. When the login to DSS succeeds, the **ServiceAccount** principal identity is created in DSSand is eligible for access to secrets.

2. Grant secret access permissions to the ServiceAccount principal.

   Log into DSS as a user with sufficient permissions to manage authorization for principals in the Kubernetes identity provider.

   Create sample secrets for the demonstration app to retrieve. The secrets **secret/hello:world1** and **secret/hello:world2** are created with arbitrary content for the purpose of this example. Create those secrets before moving on to the next step.

   For this example, we are authorizing the **default** ServiceAccount in the **secrets-safe** K8S namespace to access the secrets that exist in the DSS scope **secret/hello**. The following payload can be sent to the authorize endpoint to create this access rule:

```
{
  "principalUri": "/principal/Kubernetes/user/system.serviceaccount.secrets-safe.default",
  "resourceUri": "/secret/hello",
  "operations": [
    "read"
  ],
  "access": "allow"
}
```

Note: *The trailing part of the principal URI for the service account is* ***system.serviceaccount.secrets-safe.default*** *- Principal URIs for Kubernetes* ***ServiceAccounts*** *always take this form in DSS, that adheres to the template* ***system.serviceaccount.<namespace>.<account-name>****.*

Application Manifest Additions

The final step in demonstrating DSS secret injection into pods is to modify the manifest of the target application to include the resources that retrieve secrets and write them to the pod file system.

Shown below is an example manifest for a deployment that retrieves the secrets at paths **secret/hello:world1** and **secret/hello:world2** from an in-cluster DSS instance. The **secret-retrieval InitContainer** runs prior to the main application starting, retrieves the target secrets, and writes their contents to files on the shared volume mounted at **/run/secretssafe**.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-injection-sample
spec:
  initContainers:
    # DSS Secret Retrieval Client
  - name: secret-retrieval
    image: beyondtrust/secrets-agent:unstable
    env:
       - name: SECRETSSAFE_TARGET_SECRETS
         value: hello:world1,hello:world2
       - name: SECRETSSAFE_OUTPUT_PATH
         value: /run/secretssafe
       volumeMounts:
       - name: secrets-output
         mountPath: /run/secretssafe
  containers:
  - name: target-application
       image: busybox:1
       command: [ "sh", "-c", "--" ]
       args: [ "while true; do sleep 30; done;" ]
       volumeMounts:
       - name: secrets-output
         mountPath: /run/secretssafe
  volumes:
  - name: secrets-output
       emptyDir:
         medium: Memory
```

After saving the above pod manifest to a **.yaml** file named **secret-retrieval-example.yaml**, apply it to the cluster using **kubectl apply -f secret-retrieval-example.yaml**. This creates the pod on the cluster.

Observe the pulling of the **InitContainer** image using **kubectl describe** for the pod. Verify the target secret contents are injected to the directory at **/run/secretssafe** using **kubectl exec -it <pod-name> sh** to start an interactive shell session inside the running pod. From there, navigate to the **/run/secretssafe** directory and inspect the contents of the files **hello_world1** and **hello_world2**.

The important configuration values from the manifest above are the following environment variables set on the **secret-retrieval InitContainer**:

- **SECRETSSAFE_TARGET_SECRETS**: Comma-separated list of secrets to retrieve from DSS. Secrets are saved such that scope names become directories, and colon (**:**) delimiters are changed to underscores (**_**) in the output file name.

- **SECRETSSAFE_OUTPUT_PATH**: Path to the base directory where secret contents are placed on the pod file system. In the above example, this is defined as the path to an initially empty volume that is shared between the **InitContainer** and target application by **volumeMount** configurations.

Detailed Manifest Example

There are more options available for configuring the **secret-retrieval InitContainer** that are required in scenarios in which any of the following are true:

- DSS is not installed in the **secrets-safe** namespace.
- The identity provider is not named **kubernetes**.
- The DSSHTTPS certificate is self-signed.

Available options are:

- **SECRETSSAFE_HOST**: Host portion of the base URL for the DSS instance being contacted. For the in-cluster use case, this defaults to **standardgateway.secrets-safe.svc.cluster.local** and should only need to be changed if DSSis running in a namespace other than **secrets-safe**. Replace the **secrets-safe** segment in the URL with the DSS namespace:

  ```
  standardgateway.<DSS-namespace>.svc.cluster.local
  ```

- **SECRETSSAFE_PORT**: Port portion of the base URL for the DSS instance being contacted. For the in-cluster use case, this defaults to **8443** and should only be changed if the Gateway service of DSS is configured to use a different port. This should be set to the port number where the DSS API is exposed.
- **SECRETSSAFE_ID_PROVIDER**: The name of the Kubernetes-type identity provider configured in the DSS instance being contacted. Defaults to **kubernetes**. Change this only if the Kubernetes identity provider in the DSS instance being contacted has a name other than **kubernetes**.
- **SECRETSSAFE_VERIFY_CA**: The file path to the PEM-encoded CA certificate that can be used to verify the self-signed DSS HTTPS certificate, or the string **false** to skip certificate verification when contacting DSS.

> ***Note:** Disabling CA verification by setting this value to **false** is for testing or debugging purposes only and is not secure.*

The **InitContainer** environment variable list from the manifest example above can be expanded to include these options:

```
initContainers:
  # DSS Secret Retrieval Client
- name: secrets-agent
  image: beyondtrust/secrets-agent:unstable
  env:
  - name: SECRETSSAFE_TARGET_SECRETS
      value: hello:world1,hello:world2
  - name: SECRETSSAFE_OUTPUT_PATH
      value: /run/secretssafe
  - name: SECRETSSAFE_HOST
      value: standardgateway.<DSS-namespace>.svc.cluster.local
  - name: SECRETSSAFE_PORT
      value: 8443
  - name: SECRETSSAFE_ID_PROVIDER
      value: kubernetes
```

```
- name: SECRETSSAFE_VERIFY_CA
     value: <path-to-CA-cert-file.pem>
```

Puppet Integration

The Secrets Safe module consists of a number of plugins that allow creation and retrieval of secrets in DevOps Secrets Safe.

Setup Requirements

The functions in this module requre a running instance of DSS and an application with permissions to perform read and write permissions on the resources you interact with.

Usage

Ensure a user exists with a password retrieved from DSS, using the example shown, where **bob** is the user:

```
$user_password = dss_get_secret('https://my-secrets-safe.com', 'user/passwords:bob', "my_
application", "my_api_key")
user { 'bob':
  ensure   => present,
  password => Sensitive($user_password)
}
```

Use a Secrets Safe generator to generate a password, and then provision a Postgres database using it:

```
class { 'postgresql::server':
}

dss_create_secret_with_generator('https://my-secrets-safe.com', 'passwords/db/pg_user', "my_
application", "my_api_key", "postgres-password-generator")
$pg_pass = dss_get_secret('https://my-secrets-safe.com', 'passwords/db/pg_user', "my_application",
"my_api_key")
postgresql::server::db { 'new_postgres':
  user     => 'pg_user',
  password => postgresql::postgresql_password('pg_user', $pg_pass),
}
```

Save a certificate that is on the file system as a secret in DSS:

```
dss_create_secret_with_file('https://my-secrets-safe.com', 'certs:mycert', "my_application", "my_
api_key", "//etc/ssl/certs/ca.crt")
```

Functions

Common Parameters

Each of the following functions have some common parameters:

host	Data type: String	Hostname or IP address of Secrets Safe instance
app_name	Data type: String	Name of Secrets Safe application used to perform this action
api_key	Data type: String	API key of the Secrets Safe application specified in the **app_name** parameter
secret_uri	Data type: String	URI of the secret being operated on
secret_value	Data type: String	String value of the secret to be stored
generator_name	Data type: String	Name of the Secrets Safe generator used to generate the value for this secret
file_name	Data type: String	Path to the file which will be stored as a secret

```
dss_get_secret(host, secret_uri, app_name, api_key)
```

Returns the value of a Secrets Safe secret found at **secret_uri**.

```
dss_create_secret_with_value (host, secret_uri, app_name, api_key, secret_value)
```

Creates a secret at **secret_uri** using the value of **secret_value**.

```
dss_create_secret_with_generator(host, secret_uri, app_name, api_key, generator_name)
```

Creates a secret at **secret_uri** using the Secrets Safe generator specified in **generator_name**.

```
dss_create_secret_with_file(host, secret_uri, app_name, api_key, file_name)
```

Creates a secret at **secret_uri** using the file at **file_name** as the value.

Identity Provider Configuration

Identity providers in DevOps Secrets Safe are responsible for performing authentication and assigning identity to authenticated users. Only the internal identity provider is enabled by default. External identity providers can be configured to enable usage of identity sources separate from the internal user store.

Manage Identity Providers

Identity providers can be configured using the CLI or the API. Management permissions for identity provider configurations are **CRUD** operations the resource path /principal. Once configured, the base resource path for an identity provider is **/principal/<providerName>**. The internal identity provider exists at the path **/principal/internal**.

Users can attempt authentication via the provider using the route **/connect/token/<providerName>**. For example, if a provider were configured with the name "developers", principals from that provider would exist under the path **principal/developers** while users from that provider could log in by supplying their credentials in a request to the route **/connect/token/developers**.

Principals are created for external users the first time they successfully log in. It is not currently possible to set up permissions for specific users from external identity providers until they first perform a login. The act of logging in makes DevOps Secrets Safe aware of the user identity and makes the identity eligible for permission management.

List Identity Provider Configurations

```
ssrun identity get
```

This command returns a JSON array of all external identity provider configurations.

Example: *Output*

```
$ ssrun list-identity-providers
[
 {
   "Name": "ldap_production",
   "Type": "LDAP",
   "Options": {
     "Url": "ldap://ldap.bt.test",
     "BindDn": "uid=tesla,dc=example,dc=com",
     "BindCredentials": "password",
     "SearchBase": "dc=example,dc=com",
     "SearchFilter": "(&(objectClass=person)(uid={0}))",
     "GroupDn": "dc=example,dc=com",
     "GroupFilter": "(|(memberUid={0})(member={0})(uniqueMember={0}))"
   }
 },
 {
     "Type": "idcs_sample",
     "Name": "IDCS",
     "Options": {
         "ClientId": "abcdefg",
         "ClientSecret": "987654321",
         "InstanceUrl": "https://<siteinstance>.oraclecloud.com"
     }
 }
]
```

Create Identity Provider Configuration

```
ssrun identity create -f myConfig.txt
```

Creates the identity provider described in the file at **myConfig.txt**.

```
ssrun identity create -n <providerName>
```

Creates a pre-configured identity provider identified by the **-n** argument. Currently the only pre-configured identity provider is Kubernetes, which will configure the cluster hosting secrets-safe as an identity provider.

The pre-configured Kubernetes provider configuration is:

```
{
        "Name": "Kubernetes",
        "Type": "Kubernetes",
        "Enabled": true
}
```

For more information about valid configuration samples, please see Supported Identity Provider Types.

Update Identity Provider Configuration

```
ssrun identity update -f myConfig.txt -n <providerName>
```

Updates the identity provider named **<providerName>** with the contents of the configuration file **myConfig.txt**. The name field for a provider is static and cannot be changed by an update operation. All other fields are eligible for modification.

Delete Identity Provider Configuration

```
ssrun identity delete -n <providerName>
```

Deletes the identity provider configuration named **<providerName>**. After deletion, the named provider is erased and can no longer be used for authentication. Users whose identities originate from the deleted provider will not be able to obtain new authorization tokens.

Group Membership Synchronization for External Identity Providers

DevOps Secrets Safe supports synchronization of group membership for users and groups defined in external providers.

In order for an externally-defined group to become eligible for membership synchronization, a matching representation of the group must be created in DSS using the group management API. A unique ID for the group in DSS must be provided in the group creation call and must match the unique ID for the corresponding group in the external provider.

Group membership for external users is synchronized at login-time. Users are added to and removed from groups in DevOps Secrets Safe according to the membership lists queried from the external provider at the time the user logs in.

Examples of typical group synchronization workflows for each identity provider type are described in the provider configuration description sections below.

Supported Identity Provider Types

The supported identity provider types and their required configuration fields are listed in this section. All provider configurations require the following top-level items:

- **Name** - The name for the provider
- **Type** - The provider type (currently either "IDCS", "LDAP" or "Kubernetes")

The configuration options specific to each provider type are described in the subsections that follow.

LDAP

Sample LDAP identity provider configuration:

```
{
  "Name": "ldap_production",
  "Type": "LDAP",
  "Options": {
    "Url": "ldap://ldap.bt.test",
```

```
    "BindDn": "uid=tesla,dc=example,dc=com",
    "BindCredentials": "password",
    "SearchBase": "dc=example,dc=com",
    "SearchFilter": "(&(objectClass=person)(uid={0}))",
    "GroupDn": "dc=example,dc=com",
    "GroupFilter": "(|(memberUid={0})(member={0})(uniqueMember={0}))"
  }
}
```

Options for LDAP:

- **Url** (string, required) - URL for the target LDAP server.
  - Example: **ldap://secretssafe.test:389**, **ldaps://secretssafe.test:636**
- **Certificate** (string, optional) - CA certificate to use for verifying LDAP server certificate, must be x509 PEM-encoded.
- **StartTls** (bool, optional) - If true, issue STARTTLS request after connection to establish TLS-secure communication on an otherwise clear-text LDAP connection.
- **InsecureTls** (bool, optional) - If true, skip LDAP server SSL certificate verification - this is not secure and not recommended for production use.
- **BindDn** (string, required) - Distinguished name for target bind object when performing user and group search.
- **BindCredentials** (string, required) - Password to use with BindDn.
  - Example: **cn=admin,dc=secretssafe,dc=test**
- **SearchBase** (string, required) - Base DN for user search.
  - Example: **ou=users,dc=secretssafe,dc=test**
- **SearchFilter** (string, required) - Filter for user search. Username is inserted at the template position **{0}** from the string.
  - Example: **(&(objectClass=person)(uid={0}))**
- **GroupDn** (string, required) - Base DN under which to perform group search.
  - Example: **ou=groups,dc=secretssafe,dc=test**
- **GroupFilter** (string, optional) - Filter for group membership query. Username is inserted at the template position **{0}** from the string.
  - Defaults to: **(|(memberUid={0})(member={0})(uniqueMember={0}))**

Group Membership Synchronization

Group membership synchronization for LDAP uses the group object's DN (Distinguished Name) as the identitfier for matching local groups to groups defined on the server. DevOps Secrets Safe queries the LDAP server using the provided **GroupDn** as the search base to return a collection of Group objects. The membership list for each group is then filtered using the provided **GroupFilter** to determine which groups the currently logging-in user is a member of.

Consider the following scenario:

An LDAP identity provider is configured in DevOps Secrets Safe with the name **LDAP**. A user **tesla** exists in the remote LDAP server, with DN: **uid=tesla, ou=people, dc=secretssafe, dc=test**. The user is a member of LDAP group: **cn=scientists, ou=groups, dc=secretssafe, dc=test**.

In order to make this group's membership eligible for synchronization with DSS, we must first use the group management API to create a group with the following parameters:

UniqueID: **cn=scientists, ou=groups, dc=secretssafe, dc=test** (the group's DN according to the remote server) IdentityProvider: **LDAP** (the configured name for this identity provider in DSS)

After that group has been created in DevOps Secrets Safe, user **tesla** will be added to the group's membership list the next time they perform a login to DSS. If user **tesla** is removed from the group on the remote LDAP server, they will be removed from the corresponding DSS group at their next login.

Examples

Example: *LDAPS Scenario*

- *LDAP server running on LDAPS port 636 at* ***ldaps://ldap.secretssafe.test:636***
- *Users exist under the path* ***ou=people,dc=secretssafe,dc=test***
- *Groups exist under the path* ***ou=groups,dc=secretssafe,dc=test***
- *Bind object used for searching is* ***cn=admin,dc=secretssafe,dc=test****, with password* ***adminpass***

```
{
    "Type": "LDAP",
    "Name": "LDAP_tls",
    "Options": {
      "Url": "ldaps://ldap.secretssafe.test:636",
      "BindDn": "cn=admin,dc=secretssafe,dc=test",
      "BindCredentials": "adminpass",
      "SearchBase": "ou=people,dc=secretssafe,dc=test",
      "SearchFilter": "(&(objectClass=person)(cn={0}))",
      "GroupDn": "ou=groups,dc=secretssafe,dc=test",
      "GroupFilter": "(|(memberUid={0})(member={0})(uniqueMember={0}))",
      "Certificate": "MIIFrTCCA5UCFEncf+v6D0ZU6W="
    }
}
```

Example: *LDAP with StartTLS Scenario*

- *Server running on standard LDAP port 389 at* ***ldaps://ldap.secretssafe.test:389***
- *Server expects* ***STARTTLS*** *command to begin encrypted communication on the standard port.*
- *Users exist under the path* ***ou=people,dc=secretssafe,dc=test***
- *Groups exist under the* ***path ou=groups,dc=secretssafe,dc=test***
- *Bind object used for searching is* ***cn=admin,dc=secretssafe,dc=test****, with password* ***adminpass***

```
{
    "Type": "LDAP",
    "Name": "LDAP_starttls",
    "Options": {
      "Url": "ldap://ldap.secretssafe.test:389",
      "BindDn": "cn=admin,dc=secretssafe,dc=test",
      "BindCredentials": "adminpass",
      "SearchBase": "ou=people,dc=secretssafe,dc=test",
      "SearchFilter": "(&(objectClass=person)(cn={0}))",
      "GroupDn": "ou=groups,dc=secretssafe,dc=test",
      "GroupFilter": "(|(memberUid={0})(member={0})(uniqueMember={0}))",
      "InsecureTls": "false",
      "StartTls": "true",
      "Certificate": "MIIFrTCCA5UC="
    }
}
```

IDCS

Sample IDCS identity provider configuration:

```
{
  "Name": "idcs_sample",
  "Type": "IDCS",
  "Options": {
    "ClientId": "abcdefg",
    "ClientSecret": "987654321",
    "InstanceUrl": "https://<siteinstance>.oraclecloud.com"
  }
}
```

Required options for IDCS are:

- **ClientId** (string, required) - IDCS client ID.
- **ClientSecret** (string, required) - IDCS client secret.
- **InstanceUrl** (string, required) - Base URL for the target IDCS instance.

Group Membership Synchronization

Group membership synchronization for IDCS uses the group object's entity ID as the identitfier for matching local groups to groups defined in the remote provider. DevOps Secrets Safe uses the following IDCS API route to query group membership for a specific user:

```
admin/v1/Users/{userId}?attributes=groups
```

Kubernetes

The Kubernetes Identity Provider enables authentication to DSS using Kubernetes Service Account Tokens. Kubernetes Service Accounts become user principals in DSS that use the Service Account JWT as their credential.

For a complete guide to configuring and using this identity provider type, please see the "Kubernetes Integration" on page 42.

There are two distinct modes of operation for this identity provider. They are:

- Using the in-cluster Kubernetes API Server as the source of identity, or,
- Using a remote Kubernetes API Server as the source of identity.

In-cluster Kubernetes API Server

Minimal configuration options are required to use the same cluster DSS is running on as the source of identity. In this mode of operation, the identity provider is able to configure itself to use the local API Server from the environment DSS is running in, so no extra configuration options are required in the provider configuration.

Below is the sample Kubernetes identity provider configuration for using the local K8S cluster API Server as the source of identity:

```
{
        "Name": "Kubernetes",
        "Type": "Kubernetes",
        "Enabled": true
}
```

In order for DSS to access the K8S TokenReview API and validate ServiceAccount tokens at runtime, the K8S ServiceAccount that DSS is using to contact the K8S API server must be granted RBAC permissions for that API. These permissions can be applied using **kubectl** like so:

```
kubectl create clusterrolebinding dss-token-review-binding --clusterrole=system:auth-delegator --
serviceaccount=secrets-safe:default
```

The command example assumes that DSS is installed in the **secrets-safe** namespace. Adjust the namespace qualifier on the **serviceaccount** portion of the command to match the namespace where DSS is running.

Remote Kubernetes API Server

To use a remote Kubernetes API Server as a source of identity, extra configuration options are required. The required options are:

- **TokenReviewerJwt** (string, required): JWT for a Service Account on the target cluster that is authorized to use that cluster's TokenReview API. DSS assumes this service account identity when interacting with the target API server to validate other Service Account JWTs.
- **KubernetesCaCert** (string, required): CA certificate that is used to verify the certificate presented by the remote API server. Typically obtainable for a cluster using **kubectl** command of the form: **kubectl config view --raw --minify --flatten -o jsonpath='{.clusters[].cluster.certificate-authority-data}' | base64 --decode**
- **KubernetesHost** (string, required): Base URL for the target Kubernetes API server.

A complete sample for a Kubernetes identity provider configuration that uses a remote K8S ApiServer as the source of identity is:

```
{
        "Name": "Kubernetes",
        "Type": "Kubernetes",
        "Enabled": true,
        "Options": {
                "TokenReviewerJwt": "eyJhbGciO... <JWT for a ServiceAccount that can access TokenReview API>"
                                "KubernetesCaCert": "MIIC5zCCAc... <CA Cert for remote K8s cluster>",
                "KubernetesHost": "https://10.200.113.58:8443"
        }
}
```

Multi-Factor Authentication Configuration

Multi-factor authentication (MFA) is supported in DevOps Secrets Safe by defining MFA configurations and then associating DevOps Secrets Safe principals with those configurations, and the corresponding identities, in remote MFA providers.

Manage MFA Configurations

Multi-factor authentication can be configured using the CLI or the API. Management permissions for MFA configurations are CRUD operations on the resource path:

```
/system/multi_factor
```

List Multi-Factor Authentication Provider Configurations

```
ssrun mfa get
```

This command returns a JSON array of all MFA provider configurations.

Example:

```
$ ssrun mfa get
[
        {
                "Type": "duo",
                "Name": "BeyondTrustDuo",
                "Options": {
                        "IntegrationKey": "my integration key",
                        "SecretKey": "my-secret-key",
                        "Host": "api-myorg.duosecurity.com"
                }
        },
        {
                "Type": "duo",
                "Name": "Secrets Safe Duo",
                "Options": {
                        "IntegrationKey": "secrets safe integration key",
                        "SecretKey": "dss-secret-key",
                        "Host": "api-dss.duosecurity.com"
                }
        }
]
```

Create Multi-Factor Authentication Provider Configuration

```
ssrun mfa create -f myConfig.json
```

This creates the MFA configuration described in the file **myConfig.json**.

For valid configuration samples, see "Supported Multi-Factor Authentication Providers" on page 62.

Update Multi-Factor Authentication Provider Configuration

```
ssrun mfa update -f updatedConfig.json -n <my_configuration_name>
```

Updates the MFA configuration with the contents of the configuration file:

```
updatedConfig.json
```

The name field for a configuration is static and cannot be changed by an update operation. All other fields are eligible for modification.

Delete Multi-Factor Authentication Provider Configuration

```
ssrun mfa delete -n <my_configuration_name>
```

Deletes the configuration named:

```
<my_configuration_name>.
```

Supported Multi-Factor Authentication Providers

This section contains configuration options and sample usages of supported MFA providers.

All provider configurations require the following top-level items:

- **Name**: The name of the configuration. This must be unique across all MFA configurations.
- **Type**: The provider type.

Note: *Currently, only Duo is supported.*

The configuration for the provider type is described below.

Duo

Example:

```
{
        "type": "duo",
        "name": "My company Duo application",
        "options": {
                "IntegrationKey": "qeetyitqrtqkjpohgdjag03?=",
                "SecretKey": "j#lfae2df$?==",
                "Host": "api-my-company.duosecurity.com"
        }
}
```

Options for Duo are:

- **Host**: (String, required). URL for the Duo applications authentication API.
- **IntegrationKey**: (String, required). The Duo application integration key to be used.
- **SecretKey**: (String, required). The Duo application secret key.

Manage Multi-Factor Authentication for Principals

The following section describes how to manage MFA configurations for principals. A principal can have one or zero MFA providers configured. Only principals of type **user** or **application** support MFA.

Assign multi-factor authentication configuration to a principal

```
ssrun mfa assign-principal -p principal/internal/user/bob -m 12752112652d3c0f21551938864e2 -c
MyDuoConfig
```

This assigns **MyDuoConfig** to the internal user Bob who has the Duo id 12752112652d3c0f21551938864e2.

Remove multi-factor authentication configuration from a principal

```
ssrun mfa remove-principal -p principal/internal/user/bob
```

This removes any multi-factor authentication for the internal user Bob.

Log In as a Principal With Multi-Factor Authentication Enabled

The following section describes how a principal with multi-factor authentication enabled can log in.

```
ssrun login -u bob -m 77552
```

The code above issues a login request for internal user Bob using the MFA passscode provided by the multi-factor authentication provider.

Provider-Specific Login Functionality

The following describes MFA login functionality that is specific to the provider type:

Duo

Authentication with Duo supports pushing notifications directly to devices that support push notifications and have the Duo application installed. To use this functionality, pass the **push** keyword as the MFA passcode on login:

```
ssrun login -u bob -m push
```

The above attempts to login as internal user Bob and send a push notification to an eligible device. At that time the user can either accept or reject the authentication using their Duo application.

Event Sinks

Event Sink Configuration

Secrets Safe supports multiple event sink providers. Event sink configuration can be modified at runtime by using the CLI.

Manage Event Sink Configurations

List Event Sink Configurations

```
ssrun event-sink get
```

This command gives you a list of configured event sinks as JSON.

Example:

```
{
    "Enabled": true,
    "IsAudit": true,
    "Level": "information",
    "Name": "kibana",
    "Options": {
        "uri": "http://elk:9200"
},
    "Type": "elasticsearch"
    "uri": "/system/event_sink/kibana"
}
```

Delete an Event Sink Configuration

```
ssrun event-sink delete -n <event-sink-name>
```

This deletes the event sink with the given name.

Creating an Event Sink Configuration

```
ssrun event-sink create -f elk.json
```

This creates an event sink configuration using the values in the file **elk.json**. Details on the structure of the configuration file are outlined in the section below.

Event Sink Configuration

Configurations are defined in JSON formatted files. Event sink configurations have the following structure:

```
{
  "name": "string",
  "enabled": bool,
  "IsAudit": bool,
  "level": "string",
  "type": "string",
  "options": {
    "additionalProp1": "string",
    "additionalProp2": "string",
    "additionalProp3": "string"
  }
}
```

Field descriptions:

Required Parameters

- **name**: (Required) Friendly name for the event sink. This is the name that you provide to **ssrun event-sink delete** if you delete the event sink later.
- **level**: (Required) This is the minimum event sink event level that this event sink configuration will process. Valid levels, in ascending order, are:
    - **verbose**
    - **debug**
    - **information**
    - **warning**
    - **error**
    - **fatal**
- **type**: (Required) The event sink provider type to use. The following are supported event sink types:
    - **console**
    - **elasticsearch**
    - **syslog**

Optional Parameters

- **enabled**: (Optional, defaults to **false**). This is a flag to enable the event sink configuration. All configurations with **enabled** set to **false** will ignore all event sink events
- **IsAudit**: (Optional, defaults to **false**). This is a flag used to instruct DevOps Secrets Safe to send audit events to this sink, in addition to logs. Auditing provides details of events in the application and can create some overhead in requests, so audit logging configurations are given their own flag.
- **options**: (Optional). This is an array of key-value pairs to provide extra arguments for the event sink configuration. Some event sink types require specific options.

For example, if you provide an event sink configuration with a level of **warning**, a log event with the level **error** will be processed by your event sink; however, an event with the level of **information** will not be processed.

> ***Note:*** *Setting the **IsAudit** field to **true** will result in this field being ignored when determining if an event sink should process an event.*

Event Sink Provider Specific Options

All event sinks have the following fields in common:

- **name**
- **enabled**
- **IsAudit**
- **level**
- **type**

> ***Note:*** *The type field is what determines what, if any, options are required.*

Console Event Sink Specific Options:

Console configuration does not require or support any additional options beyond the common fields listed above.

Syslog Event Sink Specific Options:

- **uri**: (Required). This is the URI of the syslog server the logs will be shipped to.
- **Authentication**: (Optional). The type of authentication on the syslog instance. Currently only one supported value is **certificate**.
- **Certificate**: (Required if authentication type **certificate**). Base64 encoded PKCS#12 formatted keystore used by server to authenticate client
- **ValidateServerCertificate**: (Optional). Boolean indicating special client-side certificate verification should be enforced.
- **TrustedCaCertificate**: (Required if **ValidateServerCertificate** is **true**). Base64 encoded public certificate of the certificate authority that has signed the server certificates

IMPORTANT!

*Setting **ValidateServerCertificate** to **false** will disable client-side validation.*

Example:

```
{
        "Name": "external_syslog",
        "Enabled": true,
        "IsAudit": false,
        "Level": "information",
        "Type": "syslog",
        "Options": {
                "uri": tcp://sysloghost:514",
                "Authentication": "Certificate",
                "Certificate": "SGVsbG8gY3Vy",
                "ValidateServerCertificate": true,
                "TrustedCaCertificate": "LS0tLS1CRUdJ=="
        }
}
```

Elasticsearch Event Sink Specific Options:

- **uri**: (Required). The URI of the elasticsearch instance the logs will be shipped to.
- **Authentication**: (Optional). The type of authentication on the Elasticsearch instance. Supported values are **basic** and **certificate**.
- **Username**: (Required if authentication type **basic**. Can also be used with certificate authentication but is optional). Username to use for authentication.
- **Password**: (Required only if authentication type **basic**. Can also be used with certificate authentication but is optional). Password to use for authentication
- `Certificate`: (Optional). Base64 encoded PKCS#12 formatted keystore used by server to authenticate client.
- **ValidateServerCertificate**: (Optional). Boolean indicating client-side certificate verification should be enforced.
- **TrustedCaCertificate**: (Required if **ValidateServerCertificate** is **true**). Base64 encoded public certificate of the certificate authority that signed the server certificates.

Example:

Elasticsearch Logger Configuration Using No Authentication

```
{
        "Name": "external_elasticsearch",
        "Enabled": true,
        "IsAudit": false,
        "Level": "information",
        "Type": "elasticsearch",
        "Options": {
                "uri": "https://elkhost:9200",
                "Authentication": "Certificate",
                "Username": "elastic",
                "Password": "elasticPass",
                "Certificate": "SGVsbG8gY3Vy",
                "ValidateServerCertificate": "true",
                "TrustedCaCertificate": "LS0tLS1CRUdJ=="
        }
}
```

Example:

Elasticsearch Logger Configuration Using No Authentication

```
{
        "Name": "external_elasticsearch",
        "Enabled": true,
        "IsAudit": false,
        "Level": "information",
        "Type": "elasticsearch",
        "Options": {
                "uri": "http://elkhost:9200"
        }
}
```

Secret Generation

Secret Generation Configuration

DevOps Secrets Safe implements a number of secret generators. Secret generation configurations can be modified at runtime by using the CLI.

Manage Secret Generator Configurations

List Secret Generator Configurations

```
ssrun generator get
```

This provides a list of all configured secret generators as JSON.

Example:

```
{
        "Type": "String",
        "Name": "my-password-generator",
        "Description": "Default password construction policy",
        "Options": {
                "MinCharacters": 8,
                "MaxCharacters": 10,
                "AllowUpperCaseCharacters": true,
                "NumberOfRequiredUpperCaseCharacters": 1,
                "UpperCaseCharacters": "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                "AllowLowerCaseCharacters": true,
                "NumberOfRequiredLowerCaseCharacters": 1,
                "LowerCaseCharacters": "abcdefghijklmnopqrstuvwxyz",
                "AllowNumericCharacters": true,
                "NumberOfRequiredNumericCharacters": 1,
                "NumericCharacters": "1234567890",
                "AllowNonAlphaNumericCharacters": false,
                "NumberOfNonAlphaNumericCharacters": 1,
                "NonAlphaNumericCharacters": "~!@#$%^&*()-+=?/<>|[]{}_.",
                "FirstCharacterRequirement": "AnyCharacterPermitted"
        }
}
```

Note: *To view a specific generator, you can specify the generator name with the command above (**ssrun generator get -n my-password-generator**).*

Delete a Secret Generator Configuration

```
ssrun generator delete -n <generator-name>
```

This deletes the generator with the given name.

Create a Secret Generator Configuration

```
ssrun generator create -f my-generator.json
```

This creates a generator configuration using the values in the file **my-generator.json**. Details on the structure of the configuration file are outlined in the section below.

Secret Generator Configuration

Configurations are defined in JSON formatted files. Generator configurations have the following structure:

```
{
        "type": "",
        "name": "",
        "version": "1.0",
        "description": "",
        "options": {
                "option1": "",
                "option2": "",
                }
}
```

Field Descriptions

Type: (Required). The generator type to use. The following are supported generator types:

- String
- Number

These are elaborated on in a section below.

Version: (Optional). The version of the specified type to be used.

> ***Note:*** *If no version is specified, this will default to 1.0.*

Name: (Required). Friendly name for the generator. This is the name that you would provide to **ssrun generator delete** if you were to delete the secret generator later.

> ***Note:*** *Names must be unique and can only include the following characters: 0-9, A-Z, a-z, underscore (_) and dash (-).*

Description: (Optional). Provides details about this generator.

Options: (Optional). This is an array of key-value pairs to provide extra arguments for the generator configuration.

Note: *If this section or a child of this section is excluded, it will be set to the default value(s) defined by the generator type or version specifed.*

Secret Generator Provider Specific Options

As noted above, the type and version fields are what determines what, if any, options are required.

String Generator Options

The following are the options for version 1.0 of the **String** generator:

***MinCharacters:** (Defaults to **8**)	Defines the minimum password length.
***MaxCharacters:** (Defaults to **10**)	Defines the maximum password length. **MaxCharacters** must be greater than **MinCharacters**.
***AllowUpperCaseCharacters:** (Defaults to **true**)	Determines whether uppercase characters are permitted.
***AllowUpperCaseCharacters:** (Defaults to **true**)	Minimum number of required uppercase characters.
***UpperCaseCharacters:** (Defaults to **ABCDEFGHIJKLMNOPQRSTUVWXYZ**)	Defines the allowable uppercase characters.
***AllowLowerCaseCharacters:** (Defaults to **true**)	Determines whether lowercase characters are permitted.
***NumberOfRequiredLowerCaseCharacters:** (Defaults to **1**)	Minimum number of required lowercase characters.
***LowerCaseCharacters:** (Defaults to **abcdefghijklmnopqrstuvwxyz**)	Defines the allowable lowercase characters.
***AllowNumericCharacters:** (Defaults to **true**)	Determines whether numeric characters are permitted.
***NumberOfRequiredNumericCharacters:** (Defaults to **1**)	Minimum number of required numeric characters.
***NumericCharacters:** (Defaults to **1234567890**)	Defines the allowable numberic characters.
***AllowNonAlphaNumericCharacters:** (Defaults to **false**)	Determines whether non-alphanumeric characters are permitted.
***NumberOfNonAlphaNumericCharacters:** (Defaults to **1**)	Minimum number of required non-alphanumeric characters.
***NonAlphaNumericCharacters:** (Defaults to **~!@#$%^&()-+=?/<>|[]{}_.)**	Defines the allowable non-alphanumberic characters.
***FirstCharacterRequirement:** (Defaults to **AnyCharacterPermitted**)	First character value. Allowable options are: • **AnyCharacterPermitted** • **AlphaCharactersOnly** • **AlphaNumericPermitted**

Example:

```
{
        "Type": "String",
        "Name": "my-password-generator",
        "Description": "Default password construction policy",
        "Options": {
                "MinCharacters": 8,
                "MaxCharacters": 10,
                "AllowUpperCaseCharacters": true,
                "NumberOfRequiredUpperCaseCharacters": 1,
                "UpperCaseCharacters": "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                "AllowLowerCaseCharacters": true,
                "NumberOfRequiredLowerCaseCharacters": 1,
                "LowerCaseCharacters": "abcdefghijklmnopqrstuvwxyz",
                "AllowNumericCharacters": true,
                "NumberOfRequiredNumericCharacters": 1,
                "NumericCharacters": "1234567890",
                "AllowNonAlphaNumericCharacters": false,
                "NumberOfNonAlphaNumericCharacters": 1,
                "NonAlphaNumericCharacters": "~!@#$%^&*()-+=?/<>|[]{}_.",
                "FirstCharacterRequirement": "AnyCharacterPermitted"
        }
}
```

Number Generator Options

The following are the options for version 1.0 of the **Number** generator:

***MinValue:** (Defaults to **1**)	Defines the inclusive lower bound of the random number returned.
***MaxValue:** (Defaults to **9007199254740991**)	Defines the exclusive upper bound of the random number returned. **MaxValue** must be greater than **MinValue**.

Example:

```
{
        "type": "Number
        "name": "my-number-generator",
        "description": "Test Random Number Generator",
        "options": {
                "MinValue": 100,
                "MaxValue": 9007199254740991
        }
}
```

Generate Values

Seed a Secret With a Generated Value

The **create** and **update** secret commands optionally accept a generator name as an input. When specified, DevOps Secrets Safe stores a value generated by the generator instead of a value specified by the user.

> ***Note:*** *Before starting this section, ensure you have initialized, unsealed, and logged into DevOps Secrets Safe as root.*

1. Create a new user.

```
ssrun user create -n generateSecretUser -pgenerateSecretUserPassword
```

2. Create a generator.

```
ssrun generator create -f my-generator.json
```

3. Authorize the new user to create and update secrets within the resource space **secret/path/to/my/secrets**.

```
ssrun authorization create -p principal/internal/user/generateSecretUser -o create,update -a
allow secret/path/to/my/secrets
```

4. Authorize the new user to create values using the newly created generator.

```
ssrun authorization create -p principal/internal/user/generateSecretUser -o create -a allow
generator/<generator-name>
```

5. Log in as the new user.

```
ssrun login -u generateSecretUser -p generateSecretUserPassword
```

6. The new user can now use the newly created generator to generate secrets within the resource space **secret/path/to/my/secrets**.

```
ssrun secret create -g <generator-name> path/to/my/secrets:mytestsecret1
```

or

```
ssrun secret update -g <generator-name> path/to/my/secrets:mytestsecret1
```

Supported Databases

The following details supported database providers and any privileges required by the DevOps Secrets Safe database user.

Postgres

DevOps Secrets Safe currently supports Postgres 11+.

Microsoft SQL Server

DevOps Secrets Safe currently supports Microsoft SQL Server 2015+.

Oracledb

DevOps Secrets Safe currently supports Oracledb 12.2+.

Licensing

DevOps Secrets Safe has the ability to run unlicensed for an evaluation period of 30 days upon being unsealed for the first time. To continue using DSS after this evaluation period, you will need to generate licensing information using a serial number provided to you.

License Data Contents

Each of the licensing commands described below will return a common data structure detailing the current state of the license of your instance of DevOps Secrets Safe

```
{
  "SerialNumber": "AAAAA-11111-BBBBB-22222-CCCCC-33333",
  "LicenseKey": "ZZZZZZZZ-YYYYYYYY-11111111-99999999-XXXXXXXX-88888888",
  "ExpiryDate": "2020-10-14T16:39:17Z",
  "IsEvaluation": false,
  "ActiveLicenseCount": 2,
  "AllowedLicenseCount": 2147483647
}
```

Each attribute describes the following:

- **SerialNumber** - The serial number that was used to license this instance of Secrets Safe.
- **LicenseKey** - License key generated using this serial number.
- **ExpiryDate** - The expiry date of this serial number.
- **IsEvaluation** - True if this serial number is an evaluation serial number. False otherwise.
- **ActiveLicenseCount** - Number of currently active instances of Secrets Safe using this serial number.
- **AllowedLicenseCount** - Number of instances of Secrets Safe allowed for this serial number.

Apply License During Unseal

A serial number can be provided to the unseal command to update your current serial number or to apply a serial number for the first time using the **-s** parameter:

```
$ ssrun unseal -f masterkey -s AFZTG-FDRJC-6WQFU-2KIHC-G44BS-EAF65
```

If internet access is not available or it is restricted from your DevOps Secrets Safe instance, you have the option of generating a license key offline to use with this command.

For more information about generating an offline license key, please see Offline Licensing.

***Note:** If an existing serial number is in place, it will be decremented for this instance of DevOps Secrets Safe before the new serial number is applied.*

Update License

A serial number can be provided to the **license update** command to update your current serial number using the **-s** parameter:

```
$ ssrun license update -s AAAAA-BBBBB-11111-22222-CCCCC-33333
```

If internet access is not available or it is restricted from your DevOps Secrets Safe instance, you have the option of generating a license key offline to use with this command.

> For more information about generating an offline license key, please see Offline Licensing.

> ***Note:*** *If an existing serial number is in place, it will be decremented for this instance of DevOps Secrets Safe before the new serial number is applied.*

View License Data

The current state of the license of your instance of Secrets Safe can be retrieved by using the **license get** command:

```
$ ssrun license get
```

You may also use the force flag **-f** to indicate that the system should request an update of the licensing information from the licensing server.

Terminate License

To terminate your existing license use the **license delete** command:

```
$ ssrun license delete
```

> ***Note:*** *This will remove all licensing data from this instance of DevOps Secrets Safe and decrement the active instance count for the associated serial number.*

Offline Licensing

If internet access is not available or it is restricted from your DevOps Secrets Safe instance, you have the option of generating a license key offline.

To generate an offline license key:

1. Access a machine that has internet access
2. Browse to the offline licensing form licensing.beyondtrust.com
3. Enter the serial number that was provided to you and submit
4. Copy the license key that was generated for your instance of DevOps Secrets Safe

Finally, use the **-l** parameter to supply the offline license key you generated with the **unseal** or **license update** commands:

Unseal

```
$ ssrun unseal -f masterkey -s AAAAA-BBBBB-11111-22222-CCCCC-33333 -l ZZZZZZZZ-YYYYYYYY-11111111-
99999999-XXXXXXXX-88888888
```

License Update

```
$ ssrun license update -s AAAAA-BBBBB-11111-22222-CCCCC-33333 -l ZZZZZZZZ-YYYYYYYY-11111111-
99999999-XXXXXXXX-88888888
```

DevOps Secrets Safe Performance

DevOps Secrets Safe architecture was designed from inception to provide flexibility and scalability. The system is made up of a series of distributed services deployed as containers using Kubernetes. By its nature, the performance of the system varies greatly depending on the environment it is running in. To provide an idea of the performance that can be expected from the system, a reference deployment was used to gather performance statistics.

Secrets Safe Test Scenario

A test scenario was created using jmeter. Jmeter used 200 threads to simultaneously iterate over a list of users. For each user in the list, an authentication was performed and a secret was retrieved. The test continued iterating over the list of users for a period of ten minutes.

The following data was loaded into DevOps Secrets Safe prior to the execution of the test. After the data was loaded the resulting database was approximately 100 MB in size.

- 20,000 secrets, each 1024 bytes in size
- 1000 local user accounts
- Access to the secrets was granted to each user

> ***Note:*** *All audit, logs, and performance telemetry generated by Secrets Safe during the test are forwarded to an external Elasticsearch instance.*

Deployment Environments

DevOps Secrets Safe was tested in both a cloud hosted as well as an on premise virtualized environment.

Azure Environment

The Azure environment consisted of the following resources:

Service	Version	VM Host Specs
Azure Kubernetes	1.14.8	• 3 x D4s_v3 • 4 vCPU • 16GB RAM • 6400 Max IOPS
Azure Database for PostgreSQL	11	• General Purpose • 4 vCores • Local Redundant

Using this configuration DevOps Secrets Safe can handle approximately 270 incoming secret requests per second or approximately 170,000 requests over a 10 minute period.

On-Premises ESXi Environment

The on-premises environment consisted of the following resources:

Service	Version	VM Host Specs
Kubernetes cluster	1.15.5	• 3 x CentOS 7.7 • 4 Cores • 16GB RAM
PostgreSQL	10.10	• 1 x CentOS 7.7 • 4 Cores • 16GB RAM

Using this configuration DevOps Secrets Safe can handle approximately 270 incoming secret request per second or approximately 170,000 request over a 10 minute period.

Audit Volume

Action	Number of Audit Events
User Authentication	3
Secret Reterival	3
Full Performance test	~500,000 (325 MB in Elasticsearch)

Conclusions

The performance of DevOps Secrets Safe's reference deployment should be enough for most small to medium-sized customers. For larger customers, some further horizontal scaling may be required. Due to DevOps Secrets Safe's underlying architecture, this is easily achievable by adding additional Kubernetes nodes and increasing the number of replicas for the appropriate services.

API Documentation

The DevOps Secrets Safe API is written according to OpenAPI standards, which enables end users to view documentation for the API using Swagger UI. A preconfigured Swagger UI is available as part of the solution (**https://<secrets-safe-ip-dns>/swagger**).

The version specific Open API specification is included along with the deployment files as **dss-openapi.json**.