BeyondTrust

DevOps Secrets Safe 20.1
Getting Started

TC:1/28/2020

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** will be unavailable until the instance is unsealed. This command will put the 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.

> For more information about how to initialize DSS, please see Initialize DevOps Secrets Safe.

1. Create a new user

   ```
   $ ssrun user create -u 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 will return 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
   ```

   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.*

> For more information on managing secrets, please see Manage Secrets and Scopes.

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 users 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. Login as root again

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

8. Delete the new user

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

Manage Applications

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

For more information about how to log into DSS as root, please see the Initialize DevOps Secrets Safe section.

1. Create a new application

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

Note: *Upon creation, an API key will be 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 will return 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****.*

For more information on managing secrets, please see the Manage Secrets and Scopes section.

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 authorize command accepts the following arguements:

- `-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` (Required) Operations authorization applies to.
    - Options are create|read|update|delete -
- `-a` (Required) Set to allow to grant authorization or deny to revoke.

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.*

6. Read the secret

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

7. Login 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.*

Manage Secrets and Scopes

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

> For more information about how to log into DSSas root, please see the Initialize DevOps Secrets Safe section.

The next example assumes there are two files called **myTestSecretData1.txt** and **myTestSecretData2.txt** containing data you want to be stored as a secret.

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. Remove a secret

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

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

6. Remove a scope

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

Note: *This will not only remove the scope but also all scopes, secrets and metadata that is a child of it.*

Manage Metadata

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

For more information about how to log into DSS as root, please see the Initialize DevOps Secrets Safe section.

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 will simply retrieve 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
```

Manage Safelists and IP Ranges

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

For more information about how to log into DSS as root, please see the Initialize DevOps Secrets Safe section.

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
```

This example assumes there are two files called **safelist1.txt** and **safelist2.txt** with the following contents:

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 will only be enforced until the defined expiry date and will allow only IP addresses in the range of 10.101.8.10 to 10.101.8.20.*

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 will never expire and will deny 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 -s safelist_1
   ```

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

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

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

   ```
   $ ssrun ip-range get -s 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 safeLists and/or ip ranges is returned.

3. Update a safelist

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

   This command will update the safelist with name **safelist_2**.

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

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 -s safelist_2 -f ipRange.txt
   ```

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

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

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 will only be enforced until the defined expiry date and will deny IP requests coming from the IP address 10.101.8.70*

5. Update an IP range of a safelist

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

   This command will update the IP range with name ip_range_2 for the safelist with name safelist_2.

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

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 will associate the safelist with name **safelist_2** to the user with name **user1**.

7. Delete an IP range from a safelist

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

   This command will delete the IP range with name **ip_range_2** from the safelist with name **safelist_2**

8. Delete a safelist

   `$ ssrun safelist delete -s safelist_2`

   This command will delete the safelist with name **safelist_2**

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.

> Detailed instructions on event sink configuration can be found in the event sink configuration section of Event Sinks.

Install DevOps Secrets Safe

The DevOps Secrets Safe Kubernetes installation script will perform several kubectl commands to insert data into the cluster and will use Helm v2 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 in order for them to be given permission to pull the required images.

Prerequisites

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

For more information on installing Kubectl for Linux, please see Install Kubectl on Linux.

For more information on installing Helm 2, please see Install Helm.

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

***Example:** The following is an example of installing 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:** The following is an example of installing 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;'
```

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
```

> ***Note:*** *If an installation did not complete successfully, then it is recommended for the uninstaller to be run prior to the installer being run again.*

Install Certificates

The Secrets-Safe application will always serve over an HTTPS connection. By default, the standard Kubernetes self-signed certificate will be used.

If you wish to supply a custom certificate for an instance of DevOps Secrets Safe you must modify the **values.yml** and provide your certificate to the cluster before installing.

Modify the following in the **values.yml**:

1. Change the **ingress.suppliedCertificate** value to **true**
2. Change the **ingress.host** to the hostname you wish to use to refer to the ingress

In order to provide your chosen certificate to the cluster use the following command:

```
kubectl create secret tls ss-ingress-tls-secret --key ${KEY_FILE} --cert ${CERT_FILE}
```

If you do not wish to supply a custom certificate but you do want the feature of hostname-based routing, then you may leave the **suppliedCertificate** value as **false** but fill in the preferred ingress hostname.

Upgrade Instructions

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

Additional Notes - Helm

Currently the DevOps Secrets Safe Helm chart is compatible with version 2.X only.

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 a bare metal deployment the following command may be run:

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

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 stable/nginx-ingress --namespace kube-system --set controller.replicaCount=3 --version
v1.24.5 -n nginx-ingress --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 are not planning on using safelist enforcement, this option can be exluded.*

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, titled **secretssafe**, is installed and managed on a client machine by the Python package manager pip through a BeyondTrust supplied **.whl** file that 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
```

Set Up the Required Environment Variables

```
$ 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>
```

The DevOps Secrets Safe CLI verifies the SSL certificate presented by the DSS instance. The **SECRETSSAFE_VERIFY_CA** environment variable 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. As a convince, it is recommended that these environment variables be persisted in the shell environment. For example, storing them in a users **~/.bashrc** file similar to the following;

```
$ 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 test it is NOT recommended in a production environment.*

Execute the CLI

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

```
$ ssrun
```

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

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 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 update --add
```

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

```
$ ssrun group update --add --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 update --remove
```

You may also execute the command promptless as follows:

```
$ ssrun group update --remove --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.

Integrations

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

Ansible

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 will allow 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.*

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**.

> 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" or "LDAP")

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`

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 will give you a list of configured event sinks as JSON.

Example: *Output*

```
{
    "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 will create an event sink configuration using the values in the file **elk.json**. Details on the structure of the configuration file will be 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 would provide to ssrun event-sink delete if you were to 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 provided an event sink configuration with a level of "warning" then a log event with the level "error" would be processed by your event sink but an event with the level of "information" would not. It should be noted that 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) 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.
    - Warning: setting this to **false** will disable client-side validation.
- `TrustedCaCertificate` - (Required if **ValidateServerCertificate** is **true**) - base64 encoded public certificate of the certificate authority that has signed the server certificates

> ***Example:*** *Syslog Logger Configuration*

```
"Name": "external_syslog",
"Enabled": true,
"IsAudit": false,
"Level": "information",
"Type": "syslog",
"Options": {
        "uri": tcp://127.0.0.1:9200",
        "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 has signed the server certificates

Example: *Elasticsearch Logger Configuration*

```
{
        "Name": "external_elasticsearch",
        "Enabled": true,
        "IsAudit": false,
        "Level": "information",
        "Type": "elasticsearch",
        "Options": {
                "uri": "https://127.0.0.1:9200",
                "Authentication": "Certificate",
                "Username": "elastic",
                "Password": "elasticPass",
                "Certificate": "SGVsbG8gY3Vy",
                "ValidateServerCertificate": true,
                "TrustedCaCertificate": "LS0tLS1CRUdJ=="
        }
}
```

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.

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
```

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**.