Migrate to SAML from LDAP through the REST API

Published: 2020-02-23

Secure, single sign-on (SSO) authentication to the Command and Discover appliances is easy to configure. However, if you have configured your ExtraHop appliance for remote authentication through LDAP, TACACS +, or RADIUS, changing to SAML permanently deletes all existing remote users and their customizations, such as saved dashboards, activity maps, reports (for Command appliances), and record queries (for Explore appliances).

This guide provides a series of example scripts that show you how to safely migrate user customizations from remote users to SAML through the REST API. For each script, you must replace the script variables with information about your environment.

Important: Customizations must be saved from the appliance where remote users have created them. For example, if a remote user has a critical dashboard on a Command appliance and a Discover appliance, you must complete these procedures on both appliances for that remote user.

If you prefer to engage a turn-key solution for migration, contact your ExtraHop sales representative.

Procedure overview

Migrating to a new remote authentication method is a complex process. Be sure you understand all of the steps before you begin and be sure to schedule a maintenance window to avoid disrupting users.

Before you begin

1. Enable exception files on your Discover and Command appliances. If the ExtraHop system unexpectedly stops or restarts during the migration process, the exception file is written to disk. The exception file can help ExtraHop Support diagnose the issue that caused the failure.
2. Create a backup of your Discover and Command appliances. Backup files include all users, customizations, and shared settings. Download and store the backup file off-appliance to a local machine.

Because changing the remote authentication method on the appliance effectively deletes all remote users, you must first create a (mirrored) local user for each remote user where you can temporarily transfer customizations and sharing settings. After transferring these settings once, you must configure SAML for the appliance, and then transfer the settings a second time from the local users to the SAML users. Finally, you can delete the temporary local users from the appliance.

Here is an explanation of each step:

1. Retrieve metadata for customizations created by remote users.
2. (Optional for Explore appliance users) Save record queries created by remote users to the setup user account.
3. Retrieve remote users and user groups.
4. Create a temporary local user account for each remote user that you want to preserve.
5. Transfer customizations from remote user accounts to temporary local user accounts.
6. Configure SAML on the appliance. (All remote users and user groups are deleted.)
7. Create SAML user accounts for each remote user that was deleted. After the appliance is configured for SAML, you can create a remote account for your users before they log into the appliance for the first time.
8. Recreate local user groups that were deleted.
9. Transfer customizations and sharing settings from the temporary local user accounts to the new SAML user accounts. When your SAML users log in for the first time, their customizations will be available.
10. Delete the temporary local user accounts.

Retrieve remote user customizations

The following Python script retrieves a list of remote user customizations and associated metadata and saves the information in JSON files. Run the script once for each type of customization after replacing the variables with information from your environment.

Environmental variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Customization variables

The following variables specify a customization type. You must replace these variables each time that you run the script. For example, to retrieve dashboard metadata, specify `OBJECT_TYPE=dashboards` and `OBJECT_FILE=dashboards.json`:

OBJECT_TYPE

The type of customization metadata to retrieve. The following values are valid:

- `dashboards`
- `activitymaps`
- `reports`

OUTPUT_FILE

The name of the JSON file to save customization metadata in. Keep these files on your machine to input into scripts later in the migration.

- `dashboards.json`
- `activity_maps.json`
- `reports.json`

```
#!/usr/bin/python3

import json
import requests

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'
OBJECT_TYPE = 'dashboards'
OUTPUT_FILE = 'dashboards.json'

headers = {'Authorization': 'ExtraHop apikey=%s' % API_KEY}

# Method that retrieves metadata for every object
def getObjects():
    url = HOST + '/api/v1/' + OBJECT_TYPE
    r = requests.get(url, headers=headers, verify=False)
    return r.json()

# Method that retrieves sharing settings for a specified object ID
def getSharing(object_id):
    url = HOST + '/api/v1/' + OBJECT_TYPE + '/' + str(object_id) + '/
sharing'
    r = requests.get(url, headers=headers, verify=False)
    if r.status_code == 200:
```

```
        return r.json()
    else:
        print('Unable to retrieve sharing information for object ID ' +
 str(object_id))
        print(r.status_code)
        print(r.text)
        return None

eh_objects = getObjects()
if OBJECT_TYPE != 'reports':
    for eh_object in eh_objects:
        object_id = eh_object['id']
        eh_object['sharing'] = getSharing(object_id)

with open(OUTPUT_FILE, 'w') as outfile:
    json.dump(eh_objects, outfile)
```

Note: If the script returns an error message that the SSL certificate verification failed, make sure that a trusted certificate has been added to your appliance. Alternatively, you can add the `verify=False` option to bypass certificate verification. However, this method is not secure and is not recommended. The following code sends an HTTP GET request without certificate verification:

```
requests.get(url, headers=headers, verify=False)
```

(Explore appliance only) Save record queries

In the following steps, you will learn how to preserve record queries saved by a remote user.

Because saved queries can be accessed by all system users, you can export all saved queries to a bundle and then upload them after migrating to SAML. Imported record queries are assigned to the user that uploads the bundle. (For example, if you import queries from a bundle while logged in as the setup user, all of the queries list setup as the query owner.) After migration, remote users can view the saved record queries and save a copy for themselves.

This procedure must be completed from the Admin UI.

1. Log into the ExtraHop appliance with the setup user account.
2. Click the System Settings icon and then select **Bundles**.
3. From the Bundles page, select **New**.
4. Type a name to identify the bundle.
5. Click the arrow next to Queries in the Contents table and select the checkboxes next to the saved queries you want to export.
6. Click **OK**. The bundle appears in the table on the Bundles page.
7. Select the bundle and click **Download**. The queries are saved to a JSON file.

Next steps

After migration, upload the bundle to restore the saved record queries.

Retrieve remote users

The following Python script retrieves a list of remote users and their associated metadata and then saves the information in a JSON file named `user_map.json`. Replace the variables with information from your environment before you run the script.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Important: If an appliance includes duplicate LDAP user account names, the script will fail and list the duplicate names in the output. LDAP user account names are case sensitive, but SAML user account names are not. You must rename duplicate LDAP user account names before migrating them. For example, if you have LDAP user names `user_1` and `User_1`, you must rename one of those accounts before migrating to SAML.

```
#!/usr/bin/python3

import json
import requests
import sys

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'
OUTPUT_FILE = 'user_map.json'

headers = {'Authorization': 'ExtraHop apikey=%s' % API_KEY}

# Method that retrieves metadata for every user group
def getUsers():
    url = HOST + '/api/v1/users'
    r = requests.get(url, headers=headers)
    return r.json()

# Method that checks for duplicate usernames
def checkDuplicates(u_list):
    checked = []
    duplicates = set()
    for user in u_list:
        for c in checked:
            if user.lower() == c.lower():
                duplicates.add(user)
                duplicates.add(c)
        checked.append(user)
    s = sorted(duplicates, key=str.lower)
    return s

users = getUsers()
user_map = []
u_list = []
for user in users:
    if user['type'] != 'local':
        user['remote_username'] = user['username']
        user_map.append(user)
        u_list.append(user['username'])

duplicates = checkDuplicates(u_list)
if duplicates:
    print(Error: The following duplicate remote usernames were found:')
    for user in duplicates:
        print('    ' + user)
    print('Local and SAML user accounts cannot share usernames, regardless
 of case. Rename or delete duplicates before continuing.')
    sys.exit()
```

```
with open(OUTPUT_FILE, 'w') as outfile:
    json.dump(user_map, outfile)
```

Retrieve local user groups

The following Python script retrieves a list of local user groups and members and then saves the information in a JSON file named `user_groups.json`. Replace the variables with information from your environment before you run the script.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

```
#!/usr/bin/python3

import json
import requests

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'


OUTPUT_FILE = 'user_groups.json'
headers = {'Authorization': 'ExtraHop apikey=%s' % API_KEY}

# Method that retrieves metadata for every user group
def getGroups():
    url = HOST + '/api/v1/usergroups'
    r = requests.get(url, headers=headers)
    return r.json()

# Method that retrieves members of the specified group
def getMembers(group_id):
    url = HOST + '/api/v1/usergroups/' + str(group_id) + '/members'
    r = requests.get(url, headers=headers)
    if r.status_code == 200:
        return r.json()
    else:
        print('Unable to retrieve members of group ' + str(group_id))
        print(r.status_code)
        print(r.text)
        return None

groups = getGroups()
final_groups = []
for group in groups:
    if not group['is_remote']:
        group_id = group['id']
        group['members'] = getMembers(group_id)
        final_groups.append(group)

with open(OUTPUT_FILE, 'w') as outfile:
    json.dump(final_groups, outfile)
```

Create temporary local user accounts

The following Python script creates a temporary local user account for each remote user account on an appliance.

The script must be located in the same directory as the `user_map.json` file that contains the list of remote users. Replace the variables with information from your environment before you run the script.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Important: If the `user_map.json` file includes duplicate LDAP user account names, the script will fail and list the duplicate names in the output. LDAP user account names are case sensitive, but SAML user account names are not. You must rename duplicate LDAP user account names before migrating them. For example, if you have LDAP user names `user_1` and `User_1`, you must rename one of those accounts before migrating to SAML.

User variables

The following variables configure the temporary local user accounts.

PASSWORD

The password for temporary remote user accounts.

```
#!/usr/bin/python3

import json
import requests
import sys

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'
PASSWORD = '64IxICj6F0z51ZvCLdGS'


USER_FILE = 'user_map.json'
CREATE_TYPE = 'local'
if CREATE_TYPE == 'local':
    TYPE_TO_COPY = 'remote_username'
    TYPE_TO_CREATE = 'local_username'
    USER_TYPE = 'local'
elif CREATE_TYPE == 'saml':
    TYPE_TO_COPY = 'remote_username'
    TYPE_TO_CREATE = 'saml_username'
    USER_TYPE = 'remote'
else:
    print('Error: Please specify a valid CREATE_TYPE')

# Method that generates the name of the new user
# from the name of the user being copied
def generateName(name):
    return name + '_local'

# Method that creates a local account for the specified user
def createUser(new_name, user):
```

```
    url = HOST + '/api/v1/users'
    headers = {'Content-Type': 'application/json',
               'Accept': 'application/json',
               'Authorization': 'ExtraHop apikey=%s' % API_KEY}
    user_params = {'username': new_name,
                   'enabled': user['enabled'],
                   'name': user['name'],
                   'type': USER_TYPE}
    if CREATE_TYPE == 'local':
        user_params['password'] = PASSWORD
    r = requests.post(url, headers=headers, data=json.dumps(user_params))
    return r

# Method that checks for duplicate usernames
def checkDuplicates(u_list):
    checked = []
    duplicates = set()
    for user in u_list:
        for c in checked:
            if user.lower() == c.lower():
                duplicates.add(user)
                duplicates.add(c)
        checked.append(user)
    s = sorted(duplicates, key=str.lower)
    return s

with open(USER_FILE) as json_file:
    user_map = json.load(json_file)

u_list = []
for user in user_map:
    u_list.append(user[TYPE_TO_COPY])
duplicates = checkDuplicates(u_list)
if duplicates:
    print(Error: The following duplicate usernames were found:')
    for user in duplicates:
        print('    ' + user)
    print('Local and SAML user accounts cannot share usernames, regardless
 of case. Rename or delete duplicates before continuing.')
    sys.exit()

failed = []
for user in user_map:
    username = user[TYPE_TO_COPY]
    new_name = generateName(username)
    r = createUser(new_name, user)
    if r.status_code == 201:
        user[TYPE_TO_CREATE] = new_name
        print('Successfully created new user account for ' + username + ': '
 + new_name)
    else:
        failed.append([username, r.status_code, r.text])

if failed:
    print('')
    print('Failed to create duplicate local user accounts for the following
 users:')
    for error in failed:
        print('')
        for message in error:
            print(message)

with open(USER_FILE, 'w') as outfile:
```

```
    json.dump(user_map, outfile)
```

Transfer customizations to temporary local user accounts

The following Python script transfers customizations from remote user accounts to temporary local user accounts. Run the script once for each type of customization after replacing the variables with information from your environment. For example, if you want to preserve dashboards and activity maps, you will run the script once with the customization variables for dashboards and once with the customization variables for activity maps.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Customization variables

The following variables specify a customization type. You must replace these variables each time that you run the script. For example, to transfer dashboards ownership, specify `OBJECT_TYPE=dashboards` and `OBJECT_FILE=dashboards.json`:

OBJECT_TYPE

The type of customization to transfer. The following values are valid:

- `dashboards`
- `activitymaps`
- `reports`

OBJECT_FILE

The name of the JSON file that includes the customization metadata. These files must be located in the same directory as the Python script along with the `user_map.json` file that contains the list of remote users. The following values are valid:

- `dashboards.json`
- `activity_maps.json`
- `reports.json`

```
#!/usr/bin/python3

import json
import requests

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'
OBJECT_TYPE = 'activitymaps'
OBJECT_FILE = 'activity_maps.json'

USER_MAP_FILE = 'user_map.json'
TARGET_USER = 'local_username'
# Method that transfers object ownership to new users
def transferDash(eh_object, new_user):
    url = HOST + '/api/v1/' + OBJECT_TYPE + '/' + str(eh_object['id'])
    headers = {'Content-Type': 'application/json',
               'Accept': 'application/json',
               'Authorization': 'ExtraHop apikey=%s' % API_KEY}
```

```
    body = {'owner': new_user}
    r = requests.patch(url, headers=headers, data=json.dumps(body))
    if r.status_code == 204:
        return 'success'
    else:
        return r.json()

# Create a list of old users
old_users = []
with open(USER_MAP_FILE) as json_file:
    user_map = json.load(json_file)
    for user in user_map:
        old_users.append(user['remote_username'])

with open(OBJECT_FILE) as json_file:
    eh_objects = json.load(json_file)

# Create list of ExtraHop objects owned by users that will be deleted
to_do = []
for eh_object in eh_objects:
    try:
        if eh_object['owner'] in old_users:
            to_do.append(eh_object)
    except:
        continue
eh_objects = to_do

# Update ownership of each object
success = []
fail = []
for eh_object in eh_objects:
    userIndex = old_users.index(eh_object['owner'])
    user = user_map[userIndex]
    new_user = user[TARGET_USER]
    updated = transferDash(eh_object, new_user)
    if updated == 'success':
        success.append({'eh_object': eh_object,
                        'new_user': new_user})
    else:
        fail.append([eh_object, updated])

# Print out results of script
if success:
    print('Successfully updated ownership of the following ' + OBJECT_TYPE
 + ':')
    for update in success:
        print(update['eh_object']['name'])
        print('    New owner: ' + update['new_user'])
        print('')

if fail:
    print('Failed to update ownership of the following ' + OBJECT_TYPE +
 ':')
    for failure in fail:
        print('    ' + failure[0]['name'])
        print('    ' + str(failure[1]))
        print('')
```

Configure SAML on the appliance

Depending on your environment, configure SAML. Guides are available for both Okta and Google. After you configure SAML on your ExtraHop appliance, you are able to create accounts on the appliance for your remote users, and transfer their customizations before they log in for the first time.

Create SAML user accounts

The following Python script creates SAML user accounts for each deleted remote user account on an appliance.

The script must be located in the same directory as the `user_map.json` file that contains the list of remote users. Replace the variables with information from your environment before you run the script.

Note: Verify the required format for usernames that are entered in the Login ID field with the administrator of your Identity Provider. If the usernames do not match, the remote user will not be matched to the user created on the appliance.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Note: The script generates SAML usernames through the `generateName()` method. By default, the script creates new usernames by appending `@example.com` to the end of the remote username. You must configure the method to generate usernames according to your SAML user account naming standard. Verify how to format usernames with the administrator of your Identity Provider.

You can also specify SAML usernames in a CSV file. To configure the script to retrieve usernames from a CSV file, set the `READ_CSV_FILE` variable in the script to `True`. The CSV file must meet the following requirements:

- The CSV file must not contain a header row.
- Each row of the CSV file must contain the following two columns in the specified order:

ExtraHop username	SAML username

- The CSV file must be named `remote_to_saml.csv` and be located in the same directory as the Python script.

```
#!/usr/bin/python3

import json
import requests
import csv
import sys

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'


USER_FILE = 'user_map.json'
CREATE_TYPE = 'saml'
```

```
PASSWORD = '64IxICj6F0z51ZvCLdGS'
READ_CSV_FILE = False
CSV_FILE = 'remote_to_saml.csv'

if CREATE_TYPE == 'local':
    TYPE_TO_COPY = 'remote_username'
    TYPE_TO_CREATE = 'local_username'
    USER_TYPE = 'local'
elif CREATE_TYPE == 'saml':
    TYPE_TO_COPY = 'remote_username'
    TYPE_TO_CREATE = 'saml_username'
    USER_TYPE = 'remote'
else:
    print('Error: Please specify a valid CREATE_TYPE')

csv_mapping = {}
if READ_CSV_FILE and CREATE_TYPE == 'saml':
    with open(CSV_FILE, 'rt', encoding='ascii') as f:
        reader = csv.reader(f)
        for row in reader:
            csv_mapping[row.pop()] = row.pop()

# Method that generates the name of the new user
# from the name of the user being copied
def generateName(name):
    if csv_mapping:
        if name in csv_mapping:
            return csv_mapping[name]
        else:
            print('Error: Specified user ' + name + ' not found in ' +
 CSV_FILE)
            sys.exit()
    else:
        return name + '@extrahop.com'

# Method that creates a SAML account for the specified user
def createUser(new_name, user):
    url = HOST + '/api/v1/users'
    headers = {'Content-Type': 'application/json',
               'Accept': 'application/json',
               'Authorization': 'ExtraHop apikey=%s' % API_KEY}
    user_params = {'username': new_name,
                   'enabled': user['enabled'],
                   'name': user['name'],
                   'type': USER_TYPE}
    if CREATE_TYPE == 'local':
        user_params['password'] = PASSWORD
    r = requests.post(url, headers=headers, data=json.dumps(user_params))
    return r

with open(USER_FILE) as json_file:
    user_map = json.load(json_file)

failed = []
for user in user_map:
    username = user[TYPE_TO_COPY]
    new_name = generateName(username)
    r = createUser(new_name, user)
    if r.status_code == 201:
        user[TYPE_TO_CREATE] = new_name
        print('Successfully created new user account for ' + username + ': '
 + new_name)
    else:
        failed.append([username, r.status_code, r.text])
```

```
if failed:
    print('')
    print('Failed to create duplicate local user accounts for the following
 users:')
    for error in failed:
        print('')
        for message in error:
            print(message)

with open(USER_FILE, 'w') as outfile:
    json.dump(user_map, outfile)
```

Recreate local user groups

The following Python script restores membership for SAML users to local user groups.

The script must be located in the same directory as the `user_map.json` file that contains the list of remote users and the `user_groups.json` file that contains the list of user groups. Replace the variables with information from your environment before you run the script.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

```
#!/usr/bin/python3

import json
import requests

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'


USER_MAP_FILE = 'user_map.json'
GROUPS_FILE = 'user_groups.json'
# Method that updates membership in a user group
def updateMembers(group, members):
    url = HOST + '/api/v1/usergroups/' + group['id'] + '/members'
    headers = {'Content-Type': 'application/json',
               'Accept': 'application/json',
               'Authorization': 'ExtraHop apikey=%s' % API_KEY}
    r = requests.put(url, headers=headers, data=json.dumps(members))
    if r.status_code == 204:
        return 'success'
    else:
        return r.json

# Method that updates group member object by replacing old remote usernames
# with new SAML usernames
def getSamlNames(members, user_map, remote_users):
    users = members['users']
    for user in users:
        if user in remote_users:
            userIndex = remote_users.index(user)
            saml_name = user_map[userIndex]['saml_username']
```

```
            members['users'][saml_name] = members['users'].pop(user)
    return members

# Create a list of deleted remote users
remote_users = []
with open(USER_MAP_FILE) as json_file:
    user_map = json.load(json_file)
    for user in user_map:
        remote_users.append(user['remote_username'])

with open(GROUPS_FILE) as json_file:
    groups = json.load(json_file)

# Create list of local groups with remote users
to_do = []
for group in groups:
    try:
        members = group['members']['users']
        for member in members:
            if member in remote_users:
                to_do.append(group)
                break
    except:
        continue
groups = to_do

# Update group membership
success = []
fail = []
for group in groups:
    members = group['members']
    members = getSamlNames(members, user_map, remote_users)
    updated = updateMembers(group, members['users'])
    if updated == 'success':
        success.append({'group': group,
                        'users': members['users']})
    else:
        fail.append([group, updated])

# Print out results of script
if success:
    print('Successfully updated membership of the following groups:')
    for update in success:
        print(update['group']['name'])
        print('    Members:')
        for user in update['users']:
            print('        ' + user)
        print('')

if fail:
    print('Failed to update ownership of the following groups:')
    for failure in fail:
        print('    ' + failure[0]['name'])
        print('    ' + str(failure[1]))
        print('')

if success:
    with open(GROUPS_FILE, 'w') as outfile:
        json.dump(groups, outfile)
```

Transfer customizations to SAML user accounts

The following Python script transfers customizations from the temporary local user accounts to the SAML user accounts created on the appliance. Run the script once for each type of customization after replacing the variables with information from your environment. For example, if you want to preserve dashboards and activity maps, you will run the script once with the customization variables for dashboards and once with the customization variables for activity maps.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Customization variables

The following variables specify a customization type. You must replace these variables each time that you run the script. For example, to transfer dashboards ownership, specify `OBJECT_TYPE=dashboards` and `OBJECT_FILE=dashboards.json`:

OBJECT_TYPE

The type of customization to transfer. The following values are valid:

- `dashboards`
- `activitymaps`
- `reports`

OBJECT_FILE

The name of the JSON file that includes the customization metadata. These files must be located in the same directory as the Python script along with the `user_map.json` file that contains the list of remote users. The following values are valid:

- `dashboards.json`
- `activity_maps.json`
- `reports.json`

```
#!/usr/bin/python3

import json
import requests

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'
OBJECT_TYPE = 'activitymaps'
OBJECT_FILE = 'activity_maps.json'


USER_MAP_FILE = 'user_map.json'
TARGET_USER = 'saml_username'
# Method that transfers object ownership to new users
def transferDash(eh_object, new_user):
    url = HOST + '/api/v1/' + OBJECT_TYPE + '/' + str(eh_object['id'])
    headers = {'Content-Type': 'application/json',
               'Accept': 'application/json',
               'Authorization': 'ExtraHop apikey=%s' % API_KEY}
    body = {'owner': new_user}
    r = requests.patch(url, headers=headers, data=json.dumps(body))
```

```
    if r.status_code == 204:
        return 'success'
    else:
        return r.json()

# Create a list of old users
old_users = []
with open(USER_MAP_FILE) as json_file:
    user_map = json.load(json_file)
    for user in user_map:
        old_users.append(user['remote_username'])

with open(OBJECT_FILE) as json_file:
    eh_objects = json.load(json_file)

# Create list of ExtraHop objects owned by users that will be deleted
to_do = []
for eh_object in eh_objects:
    try:
        if eh_object['owner'] in old_users:
            to_do.append(eh_object)
    except:
        continue
eh_objects = to_do

# Update ownership of each object
success = []
fail = []
for eh_object in eh_objects:
    userIndex = old_users.index(eh_object['owner'])
    user = user_map[userIndex]
    new_user = user[TARGET_USER]
    updated = transferDash(eh_object, new_user)
    if updated == 'success':
        success.append({'eh_object': eh_object,
                        'new_user': new_user})
    else:
        fail.append([eh_object, updated])

# Print out results of script
if success:
    print('Successfully updated ownership of the following ' + OBJECT_TYPE
 + ':')
    for update in success:
        print(update['eh_object']['name'])
        print('    New owner: ' + update['new_user'])
        print('')

if fail:
    print('Failed to update ownership of the following ' + OBJECT_TYPE +
 ':')
    for failure in fail:
        print('    ' + failure[0]['name'])
        print('    ' + str(failure[1]))
        print('')
```

Transfer customization sharing settings to SAML user accounts

The following Python script transfers customization sharing settings from deleted remote user accounts to SAML user accounts. Run the script once for each type of customization after replacing the variables with information from your environment. For example, if you want to preserve shared settings for dashboards and

activity maps, you will run the script once with the customization variables for dashboards and once with the customization variables for activity maps.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Customization variables

The following variables specify a customization type. You must replace these variables each time that you run the script. For example, to transfer dashboards ownership, specify `OBJECT_TYPE=dashboards` and `OBJECT_FILE=dashboards.json`:

OBJECT_TYPE

The type of customization to transfer. The following values are valid:

- `dashboards`
- `activitymaps`
- `reports`

OBJECT_FILE

The name of the JSON file that includes the customization metadata. These files must be located in the same directory as the Python script along with the `user_map.json` file that contains the list of remote users and the `user_groups.json` file that contains the list of user groups. The following values are valid:

- `dashboards.json`
- `activity_maps.json`
- `reports.json`

```
#!/usr/bin/python3

import json
import requests
import sys

HOST = 'extrahop.example.com'
API_KEY = 'ac09e68cf6b5499697fe93d3930e41ed'
OBJECT_TYPE = 'dashboards'
OBJECT_FILE = 'dashboards.json'


USER_MAP_FILE = 'user_map.json'
# Method that checks to see if the specified object was shared with
# deleted users and if so, returns a sharing dictionary
# with the saml usernames
def sharedWithRemote(eh_object, remote_users, user_map):
    sharing = eh_object['sharing']
    updated = {'users': {}}
    if sharing != None:
        users_shared = sharing['users']
        for user in users_shared:
            if user in remote_users:
                user_index = remote_users.index(user)
                saml_name = user_map[user_index]['saml_username']
                updated['users'][saml_name] = sharing['users'][user]
```

```
    if updated['users']:
        return updated
    else:
        return None

# Method that updates sharing options for a specified object
def updateSharing(eh_object, remoteShares):
    url = HOST + '/api/v1/' + OBJECT_TYPE + '/'  + str(eh_object['id']) + '/
sharing'
    headers = {'Content-Type': 'application/json',
               'Accept': 'application/json',
               'Authorization': 'ExtraHop apikey=%s' % API_KEY}
    r = requests.patch(url, headers=headers, data=json.dumps(remoteShares))
    if r.status_code == 204:
        return 'success'
    else:
        return r.json()

# Create a list of deleted remote users
remote_users = []
with open(USER_MAP_FILE) as json_file:
    user_map = json.load(json_file)
    for user in user_map:
        remote_users.append(user['remote_username'])

# Extract object metadata from JSON file
with open(OBJECT_FILE) as json_file:
    eh_objects = json.load(json_file)

success = []
fail = []
# Restore sharing options for deleted remote users
for eh_object in eh_objects:
    remoteShares = sharedWithRemote(eh_object, remote_users, user_map)
    if remoteShares:
        updated = updateSharing(eh_object, remoteShares)
        if updated == 'success':
            success.append({'eh_object': eh_object,
                            'remoteShares': remoteShares})
        else:
            fail.append([eh_object, updated])

# Print out results of script
if success:
    print('Successfully updated sharing options the following ' +
 OBJECT_TYPE +  ':')
    for update in success:
        print(update['eh_object']['name'])
        print(update['remoteShares'])
        print('')

if fail:
    print('Failed to update ownership of the following ' + OBJECT_TYPE +
 ':')
    for failure in fail:
        print('    ' + failure[0]['name'])
        print('    ' + str(failure[1]))
        print('')
```

Delete temporary local user accounts

The following Python script deletes temporary local user accounts.

The script must be located in the same directory as the `user_map.json` file that contains the list of remote users. Replace the variables with information from your environment before you run the script.

Environment variables

The following variables enable the script to communicate with an appliance:

HOST

The IP address or hostname of the Discover or Command appliance.

APIKEY

The API key generated from the Discover or Command appliance.

Warning: This script permanently deletes user accounts. When a user is deleted, all customizations that are owned by that user are also deleted. Make sure that you have transferred customizations from local user accounts before running this script.

```
#!/usr/bin/python3

import json
import requests

HOST = 'extrahop.example.com'
API_KEY = '123456789abcdefghijklmnop'


USER_MAP_FILE = 'user_map.json'
# Method that deletes a user
def deleteUser(user):
    url = HOST + '/api/v1/users/' + user
    headers = {'Authorization': 'ExtraHop apikey=%s' % API_KEY}
    r = requests.delete(url, headers=headers)
    if r.status_code == 204:
        return 'success'
    else:
        return r.json()

# Create a list of temporary local users
local_users = []
with open(USER_MAP_FILE) as json_file:
    user_map = json.load(json_file)
    for user in user_map:
        local_users.append(user['local_username'])

# Delete temporary local user accounts
success = []
fail = []
for user in local_users:
    updated = deleteUser(user)
    if updated == 'success':
        success.append(user)
    else:
        fail.append([user, updated])

# Print out results of script
if success:
    print('Successfully deleted the following temporary local user
 accounts:')
    for update in success:
        print('    ' + update)

if fail:
    print('Failed to update ownership of the following dashboards:')
```

```
    for failure in fail:
        print('    ' + failure[0])
        print('    ' + str(failure[1]))
        print('')
```