

# Utility & Helper Scripts

This file documents the administrative and helper scripts used to manage and interact with the Ansible Secrets system.

## 1. Administrative Scripts

These scripts are used by administrators to manage the secret deployment process.

### add-secret.sh

**Purpose:** This script securely encrypts and adds a new secret to the Ansible Secrets system. It automates creating a GPG-encrypted password file, using the correct GPG passphrase from Ansible Vault. The script eliminates common errors from manual typos or hidden characters.

#### Installation:

This script is designed to be run from within the Ansible project directory.

- Create the file /opt/ansible\_secrets/add-secret.sh
- Add the source code below to the file.
- Make it executable. It should be owned by an administrator (e.g., flengyel).

```
sudo chown 'flengyel:domain users' /opt/ansible_secrets/add-secret.sh
sudo chmod 750 /opt/ansible_secrets/add-secret.sh
```

#### Source Code:

```
#!/bin/bash
#
# add-secret.sh - A script to securely encrypt and add a new secret
# to the Ansible Secrets project.
#
# This script automates the process of creating a GPG-encrypted password file,
# ensuring the correct GPG passphrase from Ansible Vault is used, which
# eliminates common errors from typos or hidden characters.
#
# Usage: ./add-secret.sh <secret_name>
# Example: ./add-secret.sh mfa_db
#
# The script will prompt for the password to be encrypted.

set -euo pipefail

# --- Configuration ---
# The root directory of your Ansible deployment project.
```

```

# The script must be run from a location that can access this path.
ANSIBLE_PROJECT_DIR="/opt/ansible_secrets"
FILES_DIR="${ANSIBLE_PROJECT_DIR}/files"
VAULT_FILE="${ANSIBLE_PROJECT_DIR}/group_vars/all/vault.yml"
VENV_PATH="${ANSIBLE_PROJECT_DIR}/venv/bin/activate"

# --- NEW: Define a temporary file and ensure it's cleaned up on exit ---
TEMP_FILE=$(mktemp /tmp/add-secret.XXXXXX)
trap 'rm -f "$TEMP_FILE"' EXIT

# --- Input Validation ---

# 1. Check if exactly one argument (the secret name) was provided.
if [[ $# -ne 1 ]]; then
    echo "Usage: $0 <secret_name>" >&2
    echo "Example: $0 oracle_db" >&2
    exit 1
fi

SECRET_NAME="$1"
OUTPUT_FILE="${FILES_DIR}/${SECRET_NAME}_secret.txt.gpg"

# 2. Check if required directories and files exist.
if [[ ! -d "$ANSIBLE_PROJECT_DIR" ]]; then
    echo "Error: Ansible project directory not found at '$ANSIBLE_PROJECT_DIR'" >&2
    exit 1
fi
if [[ ! -f "$VAULT_FILE" ]]; then
    echo "Error: Ansible Vault file not found at '$VAULT_FILE'" >&2
    exit 1
fi
if [[ ! -f "$VENV_PATH" ]]; then
    echo "Error: Python virtual environment not found at '$VENV_PATH'" >&2
    exit 1
fi

# 3. Prompt for the secret password securely (it will not be echoed to the screen).
read -sp "Enter the secret for '${SECRET_NAME}': " SECRET
echo # Print a newline for better formatting after the prompt.

if [[ -z "$SECRET" ]]; then
    echo "Error: Secret cannot be empty." >&2
    exit 1
fi

```

```

# 4. If the output file already exists, ask for confirmation to overwrite.
if [[ -f "${OUTPUT_FILE}" ]]; then
    read -p "Warning: '${OUTPUT_FILE}' already exists. Overwrite? (y/N) " -n 1 -r
    echo
    if [[ ! $REPLY =~ ^[Yy]$ ]]; then
        echo "Operation cancelled."
        exit 1
    fi
fi

# --- Main Logic ---

echo "--> Activating virtual environment..."
source "$VENV_PATH"

echo "--> Retrieving GPG passphrase securely from Ansible Vault..."
# This command is specifically crafted to get the passphrase value without
# any extra quotes or trailing newline characters.
GPG_PASSPHRASE=$(ansible-vault view "$VAULT_FILE" | grep 'app_gpg_passphrase:' | awk '{print $2}')

if [[ -z "$GPG_PASSPHRASE" ]]; then
    echo "Error: Failed to retrieve GPG passphrase from vault. Check vault password or file"
    exit 1
fi

echo "--> Encrypting new secret for '${SECRET_NAME}'..."
# We pipe the secret password directly into GPG's standard input.
# This avoids creating a temporary plaintext file on disk.
# --- MODIFIED: The --output now points to the temporary file. ---
printf '%s' "$SECRET" | gpg --batch --yes --symmetric --cipher-algo AES256 \
    --passphrase "$GPG_PASSPHRASE" \
    --output "$TEMP_FILE"

# Check if GPG command succeeded.
if [[ $? -eq 0 ]]; then
    echo " Success! Encrypted secret created in temporary file."
    # --- MODIFIED: Now we use sudo to move the file and set ownership. ---
    echo "--> Moving secret to final destination and setting permissions..."
    sudo mv "$TEMP_FILE" "$OUTPUT_FILE"
    sudo chown service_account:appsecretaccess "$OUTPUT_FILE"
    sudo chmod 640 "$OUTPUT_FILE" # <-- ADD THIS LINE
    echo "--> Permissions set to 640 (-rw-r-----)"
    echo "--> Final file at: ${OUTPUT_FILE}"
else

```

```

        echo " Error: GPG encryption failed." >&2
        exit 1
    fi

    # Deactivate the virtual environment
deactivate

    # The 'trap' command will automatically remove the temp file now
echo "--> Done."

```

## Usage

Must be run from within the Ansible project directory

```

cd /opt/ansible_secrets
./add-secret.sh new_secret_name

```

## Script logic

**Configuration** The script relies on four hardcoded path variables that define the structure of your Ansible project:

- **ANSIBLE\_PROJECT\_DIR**: Set to `/opt/ansible_secrets`.
- **FILES\_DIR**: The subdirectory for encrypted files,  `${ANSIBLE_PROJECT_DIR}/files`.
- **VAULT\_FILE**: The path to the Ansible Vault file containing the GPG passphrase,  `${ANSIBLE_PROJECT_DIR}/group_vars/all/vault.yml`.
- **VENV\_PATH**: The path to the Python virtual environment's activation script,  `${ANSIBLE_PROJECT_DIR}/venv/bin/activate`.

**Input Validation** Before performing any actions, the script runs several critical checks:

1. **Argument Count**: It verifies that exactly one argument (the secret's name) is provided.
2. **Environment Check**: It confirms that the Ansible project directory, the vault file, and the Python virtual environment all exist at their configured paths.
3. **Secure Prompt**: It securely prompts the administrator to enter the secret value, which is not displayed on the screen. It also ensures the provided secret is not empty.
4. **Overwrite Protection**: If an encrypted file for that secret name already exists, the script asks for explicit confirmation before overwriting it.

The script proceeds as follows:

1. **Activate Environment**: It activates the Python virtual environment to ensure `ansible-vault` is available.

2. **Retrieve GPG Passphrase:** It uses `ansible-vault view` to securely read the `app_gpg_passphrase` from the vault file in memory. The output is processed with `grep`, `awk`, and `tr` to isolate the passphrase itself without any extra characters or quotes.
3. **Encrypt the Secret:** It pipes the new secret directly into the `gpg` command's standard input. This is a secure practice as it avoids writing the plaintext secret to a temporary file.
  - It uses the `--passphrase` argument to provide the GPG key retrieved from the vault.
  - The encryption uses the `AES256` cipher.
  - The final encrypted content is written to a file named `<secret_name>_secret.txt.gpg` in the `files/` directory.
4. **Set Ownership:** Upon successful encryption, it uses `sudo` to set the new file's ownership to `service_account:appsecretaccess`, preparing it for deployment.
5. **Deactivate Environment:** The script deactivates the virtual environment, cleaning up the session.

#### `secure-app.sh`

**Purpose:** This script applies the standard production ownership (`service_account:appsecretaccess`) and permissions (0750) to an application script. It includes validation to ensure it is only run on `.sh` or `.py` files.

#### **Installation:**

This is a general-purpose utility and should be placed in a system-wide binary path.

- Create the file `/usr/local/bin/secure-app.sh`
- Add the source code below to the file.
- Make it executable:

```
sudo chmod 755 /usr/local/bin/secure-app.sh
```

#### **Source Code:**

```
#!/bin/bash

# --- Input Validation ---
if [[ $# -ne 1 ]]; then
    echo "Usage: $0 <path_to_script>" >&2
    exit 1
fi

SCRIPT_PATH="$1"

if [[ ! -f "$SCRIPT_PATH" ]]; then
    echo "Error: File not found at '$SCRIPT_PATH'" >&2
```

```

        exit 1
fi

extension="${SCRIPT_PATH##*.}"
case "$extension" in
    sh|py)
        echo "Valid extension (.${extension}) found. Securing script..."
        ;;
    *)
        echo "Error: Invalid file type. Script only supports '.sh' or '.py' extensions." >&2
        exit 1
        ;;
esac

# --- Main Logic ---
echo "Setting ownership to service_account:appsecretaccess on '$SCRIPT_PATH'"
sudo chown service_account:appsecretaccess "$SCRIPT_PATH"

echo "Setting permissions to 0750 on '$SCRIPT_PATH'"
sudo chmod 0750 "$SCRIPT_PATH"

echo "Done."

### Usage

# Must be run by an administrator with sudo privileges.
sudo /usr/local/bin/secure-app.sh /path/to/your/application_script.py

```

## 2. Runtime Helper Modules & Scripts

These are the scripts and modules used by your application scripts at runtime to retrieve secrets and establish connections. They are installed in `/usr/local/lib/ansible_secret_helpers/` and `/usr/local/bin/`.

### `secret_retriever.py`

**Purpose:** Provides the low-level `get_secret()` function for Python scripts to retrieve secrets. This is the foundation for the other helpers.

#### Installation:

- Create the file `/usr/local/lib/ansible_secret_helpers/secret_retriever.py`.
- Add the source code below.
- Set permissions: `sudo chmod 0640 /usr/local/lib/ansible_secret_helpers/secret_retriever.py` and ensure the parent directory has correct ownership (`service_account:appsecretaccess`)

and permissions (0750).

#### Source Code:

```
# /usr/local/lib/ansible_secret_helpers/secret_retriever.py
import os
import subprocess

SECRETS_DIR = "/opt/credential_store"
GPG_PASSPHRASE_FILE = os.path.join(SECRETS_DIR, ".gpg_passphrase")

def get_secret(secret_name: str) -> str:
    """
    Retrieves a decrypted secret for a given secret name.
    Raises RuntimeError on failure.
    """
    # Note the use of the _secret.txt.gpg suffix
    enc_file = os.path.join(SECRETS_DIR, f"{secret_name}_secret.txt.gpg")
    if not os.path.exists(enc_file):
        raise FileNotFoundError(f"Encrypted secret for '{secret_name}' not found.")

    cmd = ["gpg", "--batch", "--quiet", "--yes", "--passphrase-file", GPG_PASSPHRASE_FILE, enc_file]

    try:
        result = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True)
        return result.stdout.strip()
    except subprocess.CalledProcessError as e:
        raise RuntimeError(f"GPG decryption failed for '{secret_name}': {e.stderr}")
    except FileNotFoundError:
        raise RuntimeError("gpg command not found. Is GnuPG installed?")
```

connection\_helpers.py

**Purpose:** Provides high-level, reusable functions for establishing database and LDAP connections using secrets from the credential store. Your application scripts should prefer using these functions.

#### Installation:

- Create the file /usr/local/lib/ansible\_secret\_helpers/connection\_helpers.py.
- Add the source code below.
- Set permissions: `sudo chmod 0640 /usr/local/lib/ansible_secret_helpers/connection_helpers.py`

#### Source Code:

```
# /usr/local/lib/ansible_secret_helpers/connection_helpers.py
import sys
import ssl
```

```

import sqlalchemy
import cx_Oracle as cx
import ldap3
from ldap3 import Server, Connection, ALL, Tls # Requires ldap3 library
import secret_retriever # Imports the local module

def create_ldap_connection(ldap_server, user_secret, pswd_secret):
    """
    Retrieves credentials and establishes a secure LDAP connection.
    Returns a bound ldap3 Connection object.
    """
    oud_user = None
    oud_pswd = None
    try:
        oud_user = secret_retriever.get_secret(user_secret)
        oud_pswd = secret_retriever.get_secret(pswd_secret)

        tls = Tls(validate=ssl.CERT_NONE)
        srv = Server(ldap_server, port=636, get_info=ALL, use_ssl=True, tls=tls)
        oud = Connection(srv, user=oud_user, password=oud_pswd, auto_bind=True)

        # Clear credentials from memory immediately after use
        oud_user = None
        oud_pswd = None

        if not oud.bound:
            # Use the correct server variable in the error message
            raise ConnectionError(f'Error: cannot bind to {ldap_server}')

        # Return the connection object only on success
        return oud

    except Exception as e:
        # Properly handle exceptions and exit
        print(f"Error creating LDAP connection: {e}", file=sys.stderr)
        sys.exit(1)

def create_db_connection(dbhost, dbport, dbsid, user_secret, pswd_secret, engine_only=False):
    """
    Retrieves credentials and creates a DB engine and optionally a connection.
    - If engine_only is True, returns only the SQLAlchemy engine.
    - If engine_only is False (default), returns a tuple of (engine, connection).
    """
    db_user = None

```

```

db_pswd = None
engine = None
conn = None
try:
    db_user = secret_retriever.get_secret(user_secret)
    if not db_user:
        raise RuntimeError(f"Retrieved empty username secret for '{user_secret}'")

    db_pswd = secret_retriever.get_secret(pswd_secret)
    if not db_pswd:
        raise RuntimeError(f"Retrieved empty password for '{pswd_secret}'")

    datasourcename = cx.makedsn(dbhost, dbport, service_name=dbsid)
    connectstring = f'oracle+cx_oracle://'{db_user}:'{db_pswd}'@'{datasourcename}'

    # Clear credentials from memory immediately after use
    db_user = None
    db_pswd = None

    engine = sqlalchemy.create_engine(connectstring, max_identifier_length=128)

    # Conditional return based on the new flag
    if engine_only:
        return engine
    else:
        conn = engine.connect()
        return engine, conn

except Exception as e:
    print(f"Error creating database connection: {e}", file=sys.stderr)
    # Ensure resources are cleaned up on failure
    if conn:
        conn.close()
    if engine:
        engine.dispose()
    sys.exit(1)

def create_ntlm_connection(server_address, user_secret, pswd_secret):
    """
    Retrieves credentials and establishes an NTLM-authenticated LDAP connection.
    This is typically used for connecting to Microsoft Active Directory.

    It assumes the user_secret contains the full NTLM-formatted username (e.g., 'DOMAIN\\us
    Args:

```

```

server_address (str): The address of the domain controller (e.g., '100.74.1.219:389')
user_secret (str): The name of the secret storing the full username.
pswd_secret (str): The name of the secret storing the password.

>Returns:
    A bound ldap3 Connection object.

"""
ntlm_user = None
ntlm_pswd = None
conn = None
try:
    # Retrieve the full username (DOMAIN\user) and password from secrets
    ntlm_user = secret_retriever.get_secret(user_secret)
    ntlm_pswd = secret_retriever.get_secret(pswd_secret)

    # Define the server and create the connection object
    server = ldap3.Server(server_address, get_info=ldap3.ALL)
    conn = ldap3.Connection(server,
                           user=ntlm_user,
                           password=ntlm_pswd,
                           authentication=ldap3.NTLM,
                           auto_bind=True)

    if not conn.bound:
        raise ldap3.core.exceptions.LDAPBindError(f"NTLM bind failed for user {ntlm_user}")

    # Clear credentials from memory and return the connection
    ntlm_user = None
    ntlm_pswd = None
    return conn

except Exception as e:
    print(f"Error creating NTLM connection: {e}", file=sys.stderr)
    if conn and conn.bound:
        conn.unbind()
    sys.exit(1)

```

### get-secret.sh (for Bash scripts)

**Purpose:** Takes a secret name as an argument and prints the decrypted secret to standard output.

#### Installation:

See INSTALLATION.md for details.