Onboarding an Application to Ansible Secrets

This guide provides a checklist for developers to modify an existing Bash or Python script to use the centralized Ansible Secrets credential store. The process involves two main steps:

1. **Code Modification:** Updating the script to remove the hardcoded password and instead call the appropriate helper function to retrieve it at runtime.

2. **Set Secure Permissions:** Using the `secure-app.sh` utility to apply the final, correct ownership and permissions required for production use.

Step 1: Modify the Application Script

Locate the script you need to secure and modify its source code to fetch credentials from the central store.

For a Bash Script

1. Identify the line where the plaintext password is used.
2. Remove the plaintext password.
3. Add a command to call the reusable helper script `/usr/local/bin/get_secret.sh`, passing the name of the secret you need (e.g., oracle_db).
4. Store the result in a variable and check that the retrieval was successful.
5. Use the variable in your application logic.
6. unset the variable as soon as it is no longer needed.

Example Modification

- Before:

```bash
# Unsafe: password is hardcoded
DB_PASS='S3cureOracle!P@ss'
sqlplus myuser/"$DB_PASS"@ORCL @/path/to/query.sql
```

- After:

```bash
#!/bin/bash
# Get the Oracle password into a variable using the helper
ORACLE_PASS=$(/usr/local/bin/get_secret.sh oracle_db)
if [[ -z "$ORACLE_PASS" ]]; then
    echo "Failed to retrieve Oracle password from credential store." >&2
    exit 1
fi

# Use the retrieved password
sqlplus myuser/"$ORACLE_PASS"@ORCL @/path/to/query.sql

# Clear the password from memory

unset ORACLE_PASS
```

For a Python Script

1. Remove any hardcoded credentials.
2. Add the standard code block at the top of your script to add the helper library path (`/usr/local/lib/ansible_se` to `sys.path`.
3. Import the specific high-level function you need (e.g., `create_db_connection`) from the connection_helpers module.
4. Call the helper function, passing your connection parameters and the **names** of the required secrets.
5. Use the returned connection object to perform your tasks.
6. Ensure the connection and engine are closed in a `finally` block to guarantee resources are released.

Example Modification

- Before:

```python
# Unsafe: credentials are hardcoded
import sqlalchemy

constr = 'oracle+cx_oracle://myuser:S3cureP@ss@db.example.com:1521/ORCL'
engine = sqlalchemy.create_engine(constr)
conn = engine.connect()
# ...
```

- After:

```python
#!/usr/bin/env python3
import sys
import sqlalchemy

# --- Start: Required code block for secret retrieval ---
HELPER_LIB_PATH = "/usr/local/lib/ansible_secret_helpers"
if HELPER_LIB_PATH not in sys.path:
    sys.path.append(HELPER_LIB_PATH)
try:
    # Import the specific, high-level helper you need
    from connection_helpers import create_db_connection
except ImportError:
    print(f"CRITICAL: Could not import helper modules from {HELPER_LIB_PATH}.", file=sys.stderr)
    sys.exit(1)
# --- End: Required code block ---

engine, conn = None, None
try:
    # The helper handles retrieval and connection string construction.
    # Provide the names of the secrets for the user and password.
    engine, conn = create_db_connection(
        dbhost='db.example.com',
        dbport='1521',
        dbsid='ORCL',
        user_secret='my_oracle_user',      # Name of the user secret
        pswd_secret='my_oracle_password'   # Name of the password secret
    )

    print("Successfully connected to the database.")
    # ... use the 'conn' object for database operations ...

except Exception as e:
    print(f"An error occurred: {e}", file=sys.stderr)

finally:
    if conn:
        conn.close()
    if engine:
        engine.dispose()
```

Step 2: Set Secure Production Permissions

Once your script has been modified and tested, use the `secure-app.sh` utility to apply the standard production ownership and permissions. This script ensures the file is owned by `service_account:appsecretaccess` and has `0750` permissions.

```
# This is an example for a script named 'getNetID.sh'
# Replace with the path to your actual application script.
```

2

```
sudo /usr/local/bin/secure-app.sh /path/to/your/getNetID.sh
```

This command must be run by an administrator with sudo privileges.

Step 3: Final Testing

After setting the final permissions, perform a final test by running the application script.

To run the script interactively for testing, your user account must be a member of the appsecretaccess group.

If the script runs successfully, the onboarding process is complete. The script is now ready for its production use (e.g., being called by a cron job).

Step 4: Configuring for Automated Execution (Cron)

For scheduled tasks, the script must be run by a user with the correct permissions.

Requirements for Cronjobs

- **User:** The user running the cron job must be a member of the appsecretaccess group. This can be a designated administrator account or the service_account user.
- **Absolute Paths:** The cron environment is minimal. Always use absolute paths for all scripts and executables in your command (e.g., /usr/local/bin/get_secret.sh, /usr/bin/python3).
- **Logging:** Always redirect standard output (>) and standard error (2>&1) to a log file for debugging.

Example Cronjob Entry   The recommended practice is to have the cron job execute a single wrapper script. The wrapper script is responsible for setting up the environment and handling all logging.

The following example would be placed in the crontab of the user running the job (e.g., an administrator).

```
# Run the daily report job at 1:00 AM
0 1 * * * /path/to/your/wrapper_script.sh
```

This entry only specifies the schedule and the single command to run. All environment setup, logging, and error handling are contained within wrapper_script.sh, as demonstrated in EXAMPLE_CRON_JOB.md.

Example: Converting a Bash LDAP Script to Python

This example shows how to convert a Bash script that performs a simple LDAP query into a Python script that uses the high-level create_ldap_connection() helper.

Before: Bash Script (getNetID.sh)   This script securely retrieves credentials and uses the command-line tool ldapsearch to find a user's EMPLID.

```bash
#!/bin/bash
#
# Resolves a Login ID to an NetID by querying the LDAP directory.
# Accepts either a Login ID or an 10-digit NetID as input.

set -euo pipefail

# This function will be called automatically on script exit (due to 'trap')
# to ensure credentials are always cleared from memory.
cleanup() {
    unset ADMIN
    unset PSWD
}

# Trap ensures the cleanup function is called on EXIT, HUP, INT, QUIT, TERM signals.
```

```bash
trap cleanup EXIT HUP INT QUIT TERM

# --- Retrieve Secrets ---
ADMIN=$(/usr/local/bin/get_secret.sh svc_alpha)
if [[ -z "$ADMIN" ]]; then
    echo "Error: Failed to retrieve LDAP manager username." >&2
    exit 1
fi

PSWD=$(/usr/local/bin/get_secret.sh svc_beta)
if [[ -z "$PSWD" ]]; then
    echo "Error: Failed to retrieve LDAP manager password." >&2
    exit 1
fi


# --- Main Logic ---
MY_LDAP="ldaps://ldap.example.com:636"
BASEDN="CN=users,dc=example,dc=com"
CACERT="/etc/pki/tls/example.com.pem"

# Check if the first argument looks like an 10-digit EMPLID.
# If it does, just print it back out.
if [[ "$1" =~ ^[0-9]{10}$ ]]; then
    echo "$1"
    exit 0
fi

# If it's not an NetID, assume it's a login ID and query LDAP.
LOGIN_ID="$1"

# The ldapsearch command now uses safely quoted variables.
EMPLID_RESULT=$(LDAPTLS_CACERT=$CACERT \
    ldapsearch -LLL -x -H "$MY_LDAP" -D "$ADMIN" -w "$PSWD" -b "$BASEDN" -s sub "(uid=$LOGIN_ID)
    grep 'NetID:' | \
    sed 's/NetID: //')

# The 'unset' commands are now handled by the 'trap' and are not needed here.

echo "$EMPLID_RESULT"
```

After: Python Script (get_NetID.py)   This Python version accomplishes the same task but uses the con-nection_helpers module. It handles argument parsing with the argparse library and uses the ldap3 library for the search.

```python
#!/usr/bin/env python3

import sys
import argparse
import re

# --- Start: Required code block for secret retrieval ---
HELPER_LIB_PATH = "/usr/local/lib/ansible_secret_helpers"
if HELPER_LIB_PATH not in sys.path:
    sys.path.append(HELPER_LIB_PATH)
try:
    # Import the specific, high-level helper you need
    from connection_helpers import create_ldap_connection
except ImportError:
```

```python
        print(f"CRITICAL: Could not import helper modules from {HELPER_LIB_PATH}.", file=sys.stderr)
        sys.exit(1)
# --- End: Required code block ---


def main():
    """Main execution function"""
    parser = argparse.ArgumentParser(
        description="Resolves a Login ID to an NetID by querying the LDAP directory."
    )
    parser.add_argument("identifier", help="A Login ID or an 10-digit NetID.")
    args = parser.parse_args()

    # If the input is already an 10-digit NetID, just print it and exit.
    if re.fullmatch(r'\d{10}', args.identifier):
        print(args.identifier)
        sys.exit(0)

    login_id = args.identifier
    ldap_conn = None
    try:
        # Use the helper to establish a secure, authenticated LDAP connection.
        ldap_conn = create_ldap_connection(
            "oud.example.com", "svc_alpha", "svc_beta"
        )

        # Perform the LDAP search
        ldap_conn.search(
            search_base='CN=users,dc=example,dc=com',
            search_filter=f'(uid={login_id})',
            attributes=['NetID']
        )

        # Process the response
        if ldap_conn.response:
            NetID = ldap_conn.response[0]['attributes'].get('NetID', [None])[0]
            if NetID:
                print(NetID)

    except Exception as e:
        print(f"An error occurred during the LDAP query: {e}", file=sys.stderr)

    finally:
        # Ensure the LDAP connection is always closed.
        if ldap_conn:
            ldap_conn.unbind()


if __name__ == "__main__":
    main()
```