# Ansible Secrets: Examples and Usage Patterns

This directory contains practical examples for consuming secrets within application projects. These patterns ensure that sensitive data remains encrypted on disk and is only decrypted into memory at runtime by authorized users.

## Architectural Overview

Application scripts operate independently of the Ansible Deployment Project (/opt/ansible_secrets). They interact solely with the Runtime Secrets Directory (/opt/credential_store) via system-wide helper scripts.

| Component / Location | Purpose |
|---|---|
| **Bash Helper**/usr/local/bin/get_secret.sh | Decrypts a specific secret to stdout. |
| **Python Library**/usr/local/lib/ansible_secret_helpers | Provides programmatic secret retrieval. |
| **Credential Store**/opt/credential_store | Contains GPG-encrypted secrets and the master passphrase. |

## 1. Bash Example: GitHub Identity Loader

The load_github_identity.sh script is a reference implementation for non-interactively loading an SSH key into the ssh-agent.

This example uses **two secrets**:

- gitkey: the **SSH key filename** under ~{}/.ssh/. The script derives SSH_KEY_PATH="\protect\TU\textdollarHOME/.ssh/\protect\TU\textdollargitkey".
- gitphrase: the **SSH key passphrase** used by ssh-add.

### Features

- Secret-derived key selection: Retrieves gitkey and derives the key path under ~{}/.ssh/.
- Idempotency: Checks if the identity is already loaded before attempting decryption.
- Process Management: Automatically handles ssh-agent lifecycle and persists environment variables to ~{}/.ssh/agent.env.
- Security: Uses SSH_ASKPASS with a temporary, self-deleting helper script to feed the passphrase to ssh-add.

### Shell Integration

To automate your workflow, add the following to your ~{}/.bashrc or ~{}/.bashrc_custom:

# 1. Ensure the identity is loaded into the agent

load_github_identity.sh

# 2. Source the agent environment into the current shell

```
if [ -f "$HOME/.ssh/agent.env" ]; then
    . "$HOME/.ssh/agent.env" > /dev/null
fi
```

## 2. Python Example: Programmatic Retrieval

Python applications should use the secret_retriever module for direct integration.

### Usage Pattern

```
import os
from ansible_secret_helpers.secret_retriever import get_secret

def main():
    try:
        # Retrieve secret 'gitphrase' (matches gitphrase_secret.txt.gpg)
        git_pass = get_secret("gitphrase")

        # Application logic here...
        # print(f"Successfully retrieved secret of length: {len(git_pass)}")

        # Best Practice: Explicitly clear secret from memory when done
        del git_pass

    except Exception as e:
        print(f"Failed to retrieve secret: {e}")

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

## 3. Mandatory Security Protocols

To maintain the integrity of the Ansible Secrets system, all application scripts must adhere to the following:

1. Access Control: The executing user must be a member of the appsecretaccess group.
2. Memory Hygiene:

- Bash: Always use trap\unset\SECRET_VAR\EXIT to prevent secrets from lingering in the environment.
- Python: Minimize the scope of secret variables and avoid passing them to logging functions.

3. No Disk Persistence: Plaintext secrets must never be written to disk, including temporary files or /dev/shm.
4. Permissions: Production scripts should be owned by service_account:appsecretaccess with mode 0750.

## Deployment note for this example

For this example, ensure your deploy_secrets.yml deploys:

- gitkey_secret.txt.gpg
- gitphrase_secret.txt.gpg

These must appear in encrypted_secret_files, and the files must exist under /opt/ansible_secrets/files/ before running the playbook.

## Troubleshooting

- “Permission Denied”: Check group membership with groups. Note that group changes require a fresh login session to take effect.

- “GPG: decryption failed”: Ensure the GPG passphrase file is present at /opt/credential_store/.gpg_passphrase and readable by your group.