Devops UNIT-III

USING ANSIBLE TO MANAGE PASSWORDS, USERS, AND GROUPS

  - Users and passwords are the building blocks of identity management
- Groups allow you to manage a collection of users and control access to files, directories, and commands.

Enforcing Complex Passwords

- users decide what a strong password
- enforce complex passwords on every host that users can access.
- use code to enforce strong passwords for all users to achieve automation
- To do this, use an Ansible task to install a plug-in for ***Pluggable Authentication Modules (PAM)***, which is a user authentication framework that most Linux distributions employ
- The plug-in to provide complex passwords is called ***pam_pwquality.*** This module validates passwords based on criteria you set.

Installing libpam-pwquality

- The **pwquality** PAM module is available in the Ubuntu software repository under the name **libpam-pwquality**
- use the Ansible tasks to install and configure this package
- the goal is to automate everything and tasks provide the mechanism to carry out administrative work.
- These tasks are located in the repository Navigate to the ***ansible/*** directory and open the ***pam_pwquality.yml*** file in the editor
- This file contains two tasks:
- Install ***libpam-pwqualit***y and Configure ***pam_pwquality***
- first task uses the Ansible package module to install ***libpam-pwquality*** on the VM

```
---
- name: Install libpam-pwquality
  package:
    name: "libpam-pwquality"
    state: present
--snip--
```

- Each Ansible task should start with a name declaration that defines its goal.
- In this case, the name is Install libpam-pwquality.
- Next, the Ansible package module performs the software installation.
- The package module requires to set two parameters: **name and state**
- package name (found in the Ubuntu repository) should be libpam-pwquality, and the state should be **present**.
- To remove a package, set the state to **absent**.
- This is a good example of declarative instruction, since you are telling Ansible to make sure this package is installed

- Ansible modules perform common actions on an OS, such as enabling a firewall, managing users, or installing software.
- Ansible allows your actions to be ***idempotent***, which means you can do a specific action over and over again and the result will be the same as it was the last time you executed the action.
- Because of this, you should automate all you can! You'll save time and avoid mistakes caused by manual fatigue.
- Imagine if you had to configure 1,000 machines a day. It would be almost impossible without automation!

Configuring pam_pwquality to Enforce a Stricter Password Policy

- On a default Ubuntu system, It requires a minimum password length of six characters and executes only some basic complexity checks.
- To enforce more complexity, need to configure ***pam_pwquality*** to set a stricter password policy
- A file named ***/etc/pam.d/common-password*** handles configuration of the **pam_pwquality** module.
- This file is where the Ansible task makes the necessary changes to validate passwords.
- All you need to do is change one line in that file.
- A common way to edit a line using Ansible is with the ***lineinfile*** module, which allows you to change a line in a file or check whether a line exists

Ansible

```
--snip--
- name: Configure pam_pwquality
  lineinfile:
    path: "/etc/pam.d/common-password"
    regexp: "pam_pwquality.so"

    line: "password required pam_pwquality.so minlen=12 lcredit=-1 ucredit=-1
           dcredit=-1 ocredit=-1 retry=3 enforce_for_root"
    state: present
--snip--
```

- the task starts with a name, Configure **pam_pwquality**, that describes its intent.
- Then it tells Ansible to use the **lineinfile** module to edit the PAM password file. The lineinfile module requires the path of the file to which you want to make changes.
- In this case, it is the PAM password file ***/etc/pam.d/common-password***. Use a regular expression, or *regexp*, to find the line in the file you want to change.
- The regular expression locates the line that has **pam_pwquality.so** in it and replaces it with a new line. The replacement line parameter contains the **pwquality** configuration changes, which enforce more complexity.
- The options provided above enforce the following:
- • A minimum password length of 12 characters
- • One lowercase letter
- • One uppercase letter

Key Ansible Concepts

- • One numeric character
- • One nonalphanumeric character
- • Three retries
- • Disable root override
- Adding these requirements will strengthen Ubuntu’s default password policy.
- Any new passwords will need to meet or exceed these requirements, which will make brute-forcing user passwords a bit harder for attackers.

Linux User Types

- Linux, users come in three types: normal, system, and root.
- a *normal user* as a human account
- Every **normal user** is typically associated with a password, a group, and a username.
- a ***system user*** as a nonhuman account, such as the user Nginx runs as.
- a system user is almost identical to a normal user, but it is located in a different user ID (UID) range for compartmental reasons.
- A ***root user*** (or *superuser*) account has unrestricted access to the operating system. You can tell the root user by its UID, which is always zero.

Getting Started with the Ansible User Module

- Ansible comes with the user module, which makes managing users very easy.
- It handles all the messy details for accounts, like shells, keys, groups, and home directories. You'll use the user module to create a new user
- Open the *user_and_group.yml* file located in the *ansible/* directory.
- This file contains the following five tasks:
- 1. Ensure group *developers* exists.
- 2. Create the user *bender*.
- 3. Assign *bender* to the *developers* group.
- 4. Create a directory named *engineering*.
- 5. Create a file in the engineering directory.

- These tasks will create a group and a user, assign a user to a group, and create a shared directory and file.

```
--snip--
- name: Create the user 'bender'
  user:
    name: bender
    shell: /bin/bash
    password: $6$...(truncated)
--snip--
```

- The user module has many options, but only the name parameter is required. In this example, the name is set to bender.
- Setting a user's password at provision time can be useful, so set the optional password parameter field to a known password hash
- The password value, beginning with $6, is a cryptic hash that Linux supports.

Generating a Complex Password

- many different methods to generate a password to match the complexity you set in **pam_pwquality**.
- password hash that matches this threshold to save timea
- combination of two command line applications**, pwgen** and **mkpasswd**, to create the complex password.
- The **pwgen** command can generate secure passwords,
- the **mkpasswd** command can generate passwords using different hashing
- algorithms.
- The pwgen application is provided by the **pwgen package**,
- the mkpasswd application is provided by a package named **whois**.
- Together, these tools can generate the hash that Ansible and Linux expect.

- Linux stores password hashes in a file called ***shadow***.
- On an Ubuntu system, the password hashing algorithm is **SHA-512** by default.
- To create your own SHA-512 hash for Ansible's user module, use the commands below on an Ubuntu host:

```
$ sudo apt update
$ sudo apt install pwgen whois
$ pass=`pwgen --secure --capitalize --numerals --symbols 12 1`
$ echo $pass | mkpasswd --stdin --method=sha-512; echo $pass
```

- Since these packages are not installed by default, you'll need to install them first with the APT package manager.
- The pwgen command generates a complex password that matches what you need to satisfy pwquality and saves it into a variable called pass.
- Next, the contents of the variable pass are piped into mkpasswd to be hashed using the sha-512 algorithm.
- The final output should contain two lines.
- The first line contains the SHA-512 hash, and the second line contains the new password.
- You can take the hash string and set the password value in the user creation task to change it.

Linux Groups

- Linux groups allow you to manage multiple users on a host.
- Creating groups is also an efficient way to limit access to resources on a host.
- It is much easier to administer changes to a group than to hundreds of users individually
- Ansible task to create a group called ***developers*** that you will use to limit access to a directory and a file.

Getting Started with the Ansible Group Module

Like the user module, Ansible has a group module that can manage creating and removing groups. Compared to other Ansible modules, the group module is very minimal; it can only create or delete a group.

Open the *user_and_group.yml* file in your editor to review the group creation task. The first task in the file should look like this:

```
- name: Ensure group 'developers' exists
  group:
    name: developers
    state: present
--snip--
```

The `name` of the task states that you want to make sure a group exists. Use the `group` module to create the group. This module requires you to set the `name` parameter, which is set to `developers` here. The `state` parameter is set to `present`, so it will create the group if it does not already exist.

The group creation task is the first one in the file, and that is not by accident. You need to create the *developers* group before executing any other tasks. Tasks are run in order, so you need to make sure the group exists first. If you tried to reference the group before creating it, you would get an error message stating that the *developers* group doesn't exist, and the provisioning would fail. Understanding Ansible's task order of operations is key to performing more complex operations.

Keep the *user_and_group.yml* file open as you continue reviewing the other tasks.

Assigning a User to the Group

To add a user to a group with Ansible, you'll leverage the `user` module once again. In the *user_and_group.yml* file, locate the task that assigns *bender* to the *developers* group (the third task from the top in the file). It should look like this:

```
--snip--
- name: Assign 'bender' to the 'developers' group
  user:
    name: bender
    groups: developers
    append: yes
--snip--
```

First is the `name` of the task, which describes its intention. The `user` module appends *bender* to the *developers* group. The `groups` option can accept multiple groups in a comma-separated string. By using the `append` option, you leave *bender*'s previous groups intact and add only the *developers*. If you omit the `append` option, *bender* will be removed from all groups except its primary group and the one(s) listed in the `groups` parameter.

Creating Protected Resources

With *bender*'s group affiliation sorted out, let's visit the last two tasks in the *user_and_group.yml* file, which deal with creating a directory (*/opt/engineering/*) and a file (*/opt/engineering/private.txt*) on the VM. You'll use this directory and file to test user access for *bender* later.

With the *user_and_group.yml* file still open, locate the two tasks. Start with the directory creation task (the fourth from the top in the file), which should look like this:

```
- name: Create a directory named 'engineering'
  file:
    path: /opt/engineering
    state: directory
    mode: 0750
    group: developers
```

First, as before, set the `name` to match the task's intent. Use the `file` module to manage the directory and its attributes. The `path` parameter is where you want to create the directory. In this case, it's set to */opt/engineering/*. Since you want to create a directory, set the `state` parameter to the type of resource you want to create, which is `directory` in this example. You can use other types here, and you'll see another one when you create the file later. The `mode`, or privilege, is set to `0750`. This number allows the owner (*root*) to read, write, and execute against this directory, while the group members are allowed only to read and execute. The execute permission is needed to enter the directory and list its contents. Linux uses octal numbers (`0750`, in this case) to define permissions on files and groups. See the `chmod` man page for more information on permission modes. Finally, set

Summary

the group ownership of the directory to the *developers* group. This means only the users in the *developers* group can read or list the contents of this directory.

The last task in the *user_and_group.yml* file creates an empty file inside the */opt/engineering/* directory you just created. The task, located at the bottom of the file, should look like this:

```
- name: Create a file in the engineering directory
  file:
    path: "/opt/engineering/private.txt"
    state: touch
    mode: 0770
    group: developers
```

Set the task `name` to what you want to do on the host. Use the `file` module again to create a file and set some attributes on it. The `path`, which is required, gives the file's location on the VM. This example shows creating a file named *private.txt* inside the */opt/engineering/* directory. The `state` parameter is set to `touch`, which means to create an empty file if it does not exist. If you need to create a nonempty file, you can use the `copy` or `template` Ansible modules. See the documentation for more details. The `mode`, or privileges, is set to read, write, and execute for any user in the group (`0770`). Finally, set the `group` ownership of the file to the *developers* group.

USING ANSIBLE TO CONFIGURE SSH

- *SSH* is a protocol and tool that provides command line access to a remote host from your own machine.
- If you are managing a remote host or a fleet of remote hosts, the most common way to access them is over SSH.
- Most servers are likely to be headless, so the easiest way to access them is from a terminal.
- Since SSH opens access to a host, misconfiguration or default installations can lead to unauthorized access.

USING ANSIBLE TO CONFIGURE SSH

- how to use Ansible to secure SSH access to your VM
- This can be done by disabling password access over SSH, requiring
- public key authentication over SSH, and enabling two-factor authentication (2FA) over SSH

Understanding and Activating Public Key Authentication

Most Linux distributions use passwords to authenticate over SSH by default. Although this is okay for many setups, you should beef up security by adding another option: *public key authentication*. This method uses a key pair, consisting of a public key file and a private key file, to confirm your identity. Public key authentication is considered best practice for authenticating users over SSH because potential attackers who want to hijack a user's identity need both a copy of a user's private key and the passphrase to unlock it.

When you create an SSH session with a key, the remote host encrypts a *challenge* with your public key and sends the challenge back to you. Because you are in possession of the private key, you can decode the message and send back a response to the remote server. If the server can validate the response, it will know you are in possession of the private key and will thus confirm your identity. To learn more about the key exchange and SSH, visit *https://www.ssh.com/academy/ssh/*.

Generating a Public Key Pair

To generate a key pair, you'll use the `ssh-keygen` command line tool. This tool, usually installed on Unix hosts by default as part of the `ssh` package, generates and manages authentication key pairs for SSH. There's a good chance you already have a public key pair on your local host, but for this book, let's create a new key pair so you don't interfere with it. You'll also add a passphrase to the private key. A *passphrase* is like a password, but it's usually longer (more like a group of unrelated words than a complex stream of characters). You add it so that if your private key ever fell into the wrong hands, the bad actors would need to have your passphrase to unlock it and spoof your identity.

command to generate a new key pair

```
$ ssh-keygen -t rsa -f ~/.ssh/dftd -C dftd
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase): <passphrase>

Enter same passphrase again: <passphrase>
Your identification has been saved in /Users/bradleyd/.ssh/dftd.
Your public key has been saved in /Users/bradleyd/.ssh/dftd.pub.
```

You first instruct `ssh-keygen` to create an `rsa` key pair that has a name of `dftd` (DevOps for the Desperate). If you do not specify a name, it defaults to *id_rsa*, which might override your existing local key. The `-C` flag adds a human-readable comment to the end of the key that can help identify what the key is for. Here, it's also set to `dftd`. During execution, the command should prompt you to secure your key by adding a passphrase. Enter a strong passphrase to protect the key. Also remember to always keep your passphrase safe, because if you lose it, your key will become forever locked and you will never be able to use it for authentication again.

After you confirm the passphrase, the private key and public key files are created under your local *~./ssh/* directory.

Using Ansible to Get Your Public Key on the VM

Each user's home folder on the VM has a file called *authorized_keys*. This file contains a list of public keys the SSH server can use to authenticate that user. You'll use this file to authenticate *bender* when accessing the VM over SSH. To do this, you need to copy the local public key you just created in the previous section (*/Users/bradleyd/.ssh/dftd.pub*, in my case) and append the contents of that file to the */home/bender/.ssh/authorized_keys* file on the VM.

Open the *authorized_keys.yml* file in your favorite editor to review the Ansible task. The first thing you should notice is that this file has only one task. It should look like this:

```
- name: Set authorized key file from local user
  authorized_key:
    user: bender
    state: present
    key: "{{ lookup('file', lookup('env','HOME') + '/.ssh/dftd.pub') }}"
```

First, set the `name` of the task to identify its intent. Use the Ansible `authorized_key` module to copy your public key from the local host over to *bender* on the VM. The `authorized_key` module is quite simple and requires that you set only the `user` and `key` parameters. In this example, it copies the local public key you made earlier into `bender`'s */home/bender/.ssh/authorized_keys* file. Set the `state` to `present`, as you want to add the key and not remove it.

To get the contents of the local public key, you'll use Ansible's evaluation expansion operators ({{ }}) and a built-in Ansible function called `lookup`. The `lookup` function retrieves information from outside resources, based on the plug-in specified as its first argument. In this example, `lookup` uses the `file` plug-in to read the contents of the *~/.ssh/dftd.pub* public key file. The full path to this public key file is constructed with the

Adding Two-Factor Authentication

Security is built in layers. The more layers you have, the harder it is for an intruder to gain access. The next layer of security to add is *two-factor authentication (2FA)*, which validates a user's identity by using credentials and something that the user has, like a phone or device. The main goal of 2FA is to make it harder for someone to spoof your identity if your password or key is compromised.

Two-factor authentication relies on your providing two out of these three things: *something you know*, *something you have*, and *something you are*. Here are some examples of each:

Something you know: password or pin

Something you have: phone or hardware authentication device, such as a YubiKey

Something you are: fingerprint or voice

To enforce 2FA on your VM, you'll use some provided Ansible tasks to install another PAM module, configure the SSH server, and enable 2FA. To review the provided tasks, first open the *two_factor.yml* file in your editor. (All the Ansible files for this chapter are located in the *ansible/chapter3/* directory.) This file has seven tasks, and each task has a specific job to enable 2FA. The tasks are named as follows:

1. Install the `libpam-google-authenticator` package.
2. Copy over preconfigured `GoogleAuthenticator` config.
3. Disable password authentication for SSH.
4. Configure PAM to use `GoogleAuthenticator` for SSH logins.
5. Set `ChallengeResponseAuthentication` to `Yes`.
6. Set Authentication Methods for *bender*, *vagrant*, and *ubuntu*.
7. Insert an additional line here that reads: Restart SSH Server.