Optimise projet Ansible

RHCE EX294 (RHEL8)

Si vous poursuivez des objectifs de certification, vous pourriez trouvez ici des bases qui vous aideront pour :

- **3. Installation et configuration d'un nœud de contrôle Ansible**
  - 3.5. Gérer les parallélismes
- **6. Création des jeux et des playbooks Ansible**
  - 6.1. Utiliser des modules Ansible courants
  - 6.2. Utiliser des variables pour récupérer les résultats d'une commande exécutée
  - 6.3. Utiliser des conditions pour contrôler l'exécution des opérations
  - 6.4. Configurer la gestion des erreurs
  - 6.5. Créer des playbooks pour configurer des systèmes selon un état spécifique
- **8. Travailler avec des rôles**
  - 8.1. Créer et utiliser des rôles
  - 8.2. Télécharger et utiliser des rôles issus d'Ansible Galaxy
- **9. Utilisation des fonctions Ansible avancées**
  - 9.2. Utiliser Ansible Vault dans des playbooks pour protéger les données sensibles
  - 9.3. Utilisation de variables et de facts Ansible
- Exécuter sélectivement des tâches spécifiques dans des playbooks à l'aide de balises ("tags")

1. Gestion des Erreurs Ansible

1.1. Blocs (blocks) de tâches

Blocks

Les blocs permettent le regroupement logique des tâches et la gestion des erreurs dans le jeu. Tout ce que l'on peut appliquer à une tâche unique peut être appliqué au niveau du bloc ce qui facilite également la définition de données ou de directives communes aux tâches. Cela ne signifie pas que la directive affecte le bloc lui-même, mais elle est héritée par les tâches délimitées dans un bloc. Les directives seront appliqué aux tâches, mais pas au bloc lui-même.

```
tasks:
  - name: Install Apache
    block:
      - yum:
          name:
            - httpd
            - memcached
          state: installed
      - template:
          src: templates/src.j2
          dest: /etc/foo.conf
      - service:
          name: httpd
          state: started
          enabled: True
    when: ansible_distribution == 'CentOS'
    become: true
    become_user: root
```

Dans l'exemple ci-dessus, chacune des 3 tâches sera exécutée après l'ajout de la condition `when:` du bloc et son évaluation dans le contexte de la tâche. En outre, elles héritent des directives relatives à l'élévation des privilèges root (`become:` et `become_user:`) .

1.2. Gestion des exceptions

Les blocs permettent également de gérer les sorties d'erreurs de manière similaire aux exceptions de la plupart des langages de programmation.

```
tasks:
  - name: Attempt and graceful roll back demo
    block:
      - debug:
          msg: 'I execute normally'
      - command: /bin/false
      - debug:
          msg: 'I never execute, due to the above task failing'
    rescue:
      - debug:
          msg: 'I caught an error'
```

```
    - command: /bin/false
    - debug:
        msg: 'I also never execute :-('
  always:
    - debug:
        msg: "This always executes"
```

Les tâches du `block:` s'exécutent normalement.

En cas d'erreur, la section `rescue:` s'exécute avec tout ce que vous devez faire pour résoudre l'erreur précédente.

La section `always:` est exécutée quelle que soit l'erreur précédente qui s'est produite ou non dans les sections de `block:` et `rescue:`.

Il convient de noter que le jeu continue si une section `rescue:` s'achève avec succès car elle "efface" le statut de l'erreur (mais n'en fait pas le rapport), ce qui signifie qu'elle ne déclenchera ni les configurations `max_fail_percentage` ni `any_errors_fatal`, mais elle apparaîtra dans les statistiques du livre de jeu.

Un autre exemple de la prise en charge des erreurs.

```
tasks:
  - name: Attempt and graceful roll back demo
    block:
      - debug:
          msg: 'I execute normally'
        notify: run me even after an error
      - command: /bin/false
    rescue:
      - name: make sure all handlers run
        meta: flush_handlers
handlers:
  - name: run me even after an error
    debug:
      msg: 'This handler runs even on error'
```

Ansible fournit également quelques variables pour les tâches de la partie `rescue:` d'un bloc.

- `ansible_failed_task` : La tâche qui a renvoyé "failed" et a déclenché le sauvetage (rescue). Par exemple, pour obtenir le nom, utilisez `ansible_failed_task.name`.
- `ansible_failed_result` Le résultat de routeur capturé de la tâche ayant échoué qui a déclenché le sauvetage. On pourait obtenir cette valeur grâce au mot clé `register`.

1.3. Ignorer les tâches en échec

Une tâche qui échoue (failed) arrête la lecture du livre de jeu.

`ignore_errors` permet d'outrepasser ce comportement.

```
- name: this will not be counted as a failure
  command: /bin/false
  ignore_errors: yes
```

1.4. Contrôler l'état changed

Annulation du résultat "changed". Lorsqu'un shell, une commande ou un autre module s'exécute, il indique généralement le statut "changed" selon qu'il pense ou non qu'il affecte l'état de la machine.

En fonction du code de retour ou de la sortie, on sait que celui-ci n'a apporté aucune modification et on souhaiterait alors remplacer le résultat "changed" de sorte qu'il n'apparaisse pas dans la sortie du rapport ou ne provoque pas le déclenchement des handlers.

Il s'agit donc d'une condition qui contrôle l'état "changed" certainement en vue de rendre les livres de jeu idempotents.

```
  - name: Create DB if not exists
    ....
    register: db_create_result
    changed_when: "not db_create_result.stdout|search('already exists. Skipped')"
```

ou encore :

```
---
- hosts: web
  tasks:
    - name: "Start the Apache HTTPD Server"
      register: starthttpdout
      shell: "httpd -k start"
      changed_when: "'already running' not in starthttpdout.stdout"
    - debug:
```

```
    msg: "{{starthttpdout.stdout}}"
```

1.5. Contrôler l’état failed

De manière semblable, `failed_when` permet de contrôler l’état “failed”.

On trouvera ici un exemple de tâches qui créent un état “failed” qui en temps normal réussi toujours. Il s’agit de les utiliser comme “requirements” en espace de stockage d’un application Weblogic de Oracle.

```
---
- hosts: app
  tasks:
    - name: Making sure the /tmp has more than 1gb
      shell: "df -h /tmp|grep -v Filesystem|awk '{print $4}'|cut -d G -f1"
      register: tmpspace
      failed_when: "tmpspace.stdout|float < 1"

    - name: Making sure the /opt has more than 4gb
      shell: "df -h /opt|grep -v Filesystem|awk '{print $4}'|cut -d G -f1"
      register: tmpspace
      failed_when: "tmpspace.stdout|float < 4"

    - name: Making sure the Physical Memory more than 2gb
      shell: "cat /proc/meminfo|grep -i memtotal|awk '{print $2/1024/1024}'"
      register: memory
      failed_when: "memory.stdout|float < 2"
```

1.6. Tâche en échec et handlers

Une tâche qui échoue arrête la lecture du livre de jeu et empêche la prise en charge de “handlers” notifié par les tâches précédentes qui interviennent à la fin du jeu.

On peut placer la directive `force_handlers: True` dans le jeu pour qu’une tâche qui échoue n’empêche pas l’exécution de “handlers” notifiés.

Lorsque les “handlers” sont forcés, ils s’exécutent lorsqu’ils sont notifiés, même si une tâche échoue sur cet hôte.

1.7. Module fail

Le module fail met en échec la progression du jeu avec un message personnalisé.

```
---
- name: "End play if condition is meet"
  fail:
    msg: "End play with failed status"
  when: variable is defined
```

2. Gestion Connexions Ansible

2.1. Contrôle de l’élévatation de privilège dans le livre de jeu.

https://docs.ansible.com/ansible/latest/user_guide/become.html

```
- name: "PLAY 1: demo playbook"
  hosts: localhost
  remote_user: user
  become: yes
  become_user: root
  become_method: sudo
```

2.2. Contrôle de l’élévatation de privilège dans l’inventaire

...

2.3. Types de connexions Ansible

```
- name: "PLAY 1: demo playbook"
  hosts: localhost
  connection: local
```

...

Connection Plugins

Les plug-ins de connexion permettent à Ansible de se connecter aux hôtes cibles afin d'exécuter des tâches sur ceux-ci. Ansible est livré avec de nombreux plugins de connexion, mais un seul peut être utilisé par hôte à la fois. Les plus utilisés sont les types de connexion Paramiko SSH, ssh natif (appelé simplement ssh) et local.

On peut utiliser `ansible-doc -t connection -l` pour voir la liste des plugins disponibles. `ansible-doc -t <nom du plug-in>` permet d'afficher une documentation détaillée et des exemples.

- buildah Interact with an existing buildah container
- chroot Interact with local chroot
- docker Run tasks in docker containers
- funcd Use funcd to connect to target
- httpapi Use httpapi to run command on network appliances
- iocage Run tasks in iocage jails
- jail Run tasks in jails
- kubectl Execute tasks in pods running on Kubernetes.
- libvirt_lxc Run tasks in lxc containers via libvirt
- local execute on controller
- lxc Run tasks in lxc containers via lxc python library
- lxd Run tasks in lxc containers via lxc CLI
- netconf Provides a persistent connection using the netconf protocol
- network_cli Use network_cli to run command on network appliances
- oc Execute tasks in pods running on OpenShift.
- paramiko_ssh Run tasks via python ssh (paramiko)
- persistent Use a persistent unix socket for connection
- psrp Run tasks over Microsoft PowerShell Remoting Protocol
- saltstack Allow ansible to piggyback on salt minions
- ssh connect via ssh client binary
- winrm Run tasks over Microsoft's WinRM
- zone Run tasks in a zone instance

2.4. Actions Locales et Délégation

Delegation, Rolling Updates, and Local Actions

Si vous souhaitez exécuter une tâche sur un hôte en faisant référence à d'autres hôtes, utilisez le mot-clé `delegate_to` sur une tâche.

```
---

- hosts: webservers
  serial: 5

  tasks:

  - name: take out of load balancer pool
    command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
    delegate_to: 127.0.0.1

  - name: actual steps would go here
    yum:
      name: acme-web-stack
      state: latest

  - name: add back to load balancer pool
    local_action: command /usr/bin/add_back_to_pool {{ inventory_hostname }}
```

Il existe également une syntaxe abrégée `local_action:` qui inclut la tâche locale au lieu `delegate_to: 127.0.0.1` au niveau du module.

```
---
# ...
  tasks:

  - name: Send summary mail
    local_action:
      module: mail
      subject: "Summary Mail"
      to: "{{ mail_recipient }}"
      body: "{{ mail_body }}"
    run_once: True
```

La procédure `delegate_to:` est de toute façon celle qui sera préférée.

2.5. Parralélisme et le plug-in de stratégies.

```
ansible-doc -t strategy -l
debug       Executes tasks in interactive debug session
free        Executes tasks without waiting for all hosts
host_pinned Executes tasks on each host without interruption
linear      Executes tasks in a linear fashion
```

Avec le plug-in de stratégie `linear`, on peut utiliser la directive de jeu `serial: <nombre>` pour limiter le nombre tâche à exécuter à la fois.

...

2.6. Paramètre de fork

...

3. Abstraction des tâches

- Including and Importing
- Roles

Il est possible d' inclure ou d'importer dans un livre de jeu des fichiers qui comprennent une liste de jeux ou de tâches. L'importation est statique et l'inclusion est dynamique. Ansible Galaxy fait référence au site Web de Galaxy https://galaxy.ansible.com à partir duquel les utilisateurs peuvent partager des rôles. Il fait aussi référence à un outil en ligne de commande pour l'installation, la création et la gestion de rôles à partir de dépôts git. Les rôles permettent de charger automatiquement certains fichiers vars_files, tasks et handlers en fonction d'une structure de fichier connue. Le regroupement de contenu par rôles permet également de les partager facilement avec d'autres utilisateurs. En bref, une organisation en rôle n'est jamais qu'une manière d'abstraire son livre de jeu. Toutes les règles de conception d'un livre de jeu sont respectées sur base d'une structure de fichiers et de dossiers connue.

3.1. Inclusions

Il est possible d' "inclure" dans un livre de jeu des fichiers qui comprennent une liste de jeux ou de tâches avec un module `include*`. Notons que le module `include` est déprécié depuis la version Ansible 2.4 au profit de `include_tasks` et `import_tasks`

Le module `include_tasks` est appelé sur le bon niveau hiérarchique. Un liste de tâches se trouvera sous la directive `tasks`.

Par exemple :

```
---
#inclusions.yaml
- hosts: all
  tasks:
    - debug:
        msg: task1
    - name: Include task list in play
      include_tasks: stuff.yaml
    - debug:
        msg: task10
- hosts: all
  gather_facts: true
  tasks:
    - debug:
        msg: task1
    - name: Include task list in play only if the condition is true
      include_tasks: other_stuff.yaml
      when: hostvar is defined
```

Ce livre de jeu est constitué de deux jeux. Le premier jeu comprend trois tâches. La seconde est un "include" d'un fichier `stuff.yaml` constitué d'une liste de tâches. Le second jeu comprend deux tâches dont la dernière prend la liste de tâche d'un autre fichier `other_stuff.yaml`.

3.2. Imports

On connait aussi le module `import_tasks` dont le fonctionnement est similaire à `include_tasks`.

```
---
#imports.yaml
- hosts: all
```

```
  tasks:
    - debug:
        msg: task1
    - name: Import task list in play
      import_tasks: stuff.yaml
    - debug:
        msg: task10
```

ou encore :

```
tasks:
- import_tasks: wordpress.yml
  vars:
    wp_user: timmy
- import_tasks: wordpress.yml
  vars:
    wp_user: alice
- import_tasks: wordpress.yml
  vars:
    wp_user: bob
```

3.3. Import ou Include ?

Quelle est la différence entre `import_*` et `include_*` ?

Toutes les instructions `import_*` sont pré-traitées au moment de l'analyse des livres de jeu. Toutes les instructions `include_*` sont traitées au fur et à mesure lors de l'exécution du livre de jeu.

Autrement dit, l'importation est statique, l'inclusion est dynamique.

Sur base de l'expérience, on devrait utiliser `import` lorsque l'on traite avec des "unités" logiques. Par exemple, une longue liste de tâches dans des fichiers de sous-tâches :

main.yml:

```
- import_tasks: prepare_filesystem.yml
- import_tasks: install_prerequisites.yml
- import_tasks: install_application.yml
```

Mais on utiliserait de préférence `include` pour traiter différents flux de travail et prendre des décisions en fonction de “gathered facts” de manière dynamique :

install_prerequisites.yml:

```
- include_tasks: prerequisites_{{ ansible_os_family | lower }}.yml
```

3.4. import_playbook

Le module `import_playbook` intervient plus au niveau du livre de jeu :

```
---
# file: site.yml
- import_playbook: webservers.yml
- import_playbook: dbservers.yml
```

ou encore :

```
- hosts: localhost
  tasks:
    - debug:
        msg: play1

- name: Include a play after another play
  import_playbook: otherplays.yaml
```

3.5. Roles Ansible-Galaxy

Ansible Galaxy fait référence au site Web de Galaxy à partir duquel les utilisateurs peuvent partager des rôles. Il fait aussi référence à un outil en ligne de commande pour l’installation, la création et la gestion de rôles à partir de dépôts git.

Les rôles permettent de charger automatiquement certains fichiers `vars_files`, `tasks` et `handlers` en fonction d’une structure de fichier connue. Le regroupement de contenu par rôles permet également de les partager facilement avec d’autres utilisateurs.

En bref, une organisation en rôle n’est jamais qu’une manière d’**abstraire les tâches** d’un livre de jeu. Toutes les règles de conception d’un livre de jeu sont respectées sur base d’une structure de fichiers et de dossiers connue. On se base alors sur une structure de dossier et de fichiers ***par***

convention.

Exemple de structure d'un projet utilisant des rôles.

```
site.yml
webservers.yml
fooservers.yml
roles/
    common/               # Cette hiérarchie représente un "role"
        tasks/            #
            main.yml      #  <-- peut inclure des fichiers de tâches plus petits si cela est justifié
        handlers/         #
            main.yml      #  <-- fichiers de handlers
        templates/        #  <-- fichiers à utiliser avec le module "template"
            ntp.conf.j2   #  <------- modèles finissent en .j2
        files/            #
            bar.txt       #  <-- fichiers à utiliser avec le module "copy"
            foo.sh        #  <-- fichiers de script à utiliser avec le module "script"
        vars/             #
            main.yml      #  <-- variables associées avec le rôle
        defaults/         #
            main.yml      #  <-- variables par défaut (avec la plus basse priorité)
        meta/             #
            main.yml      #  <-- dépendances du rôle
    webservers/
        tasks/
        defaults/
        meta/
```

Les rôles s'attendent à ce que les fichiers se trouvent dans certains répertoires dont le nom est connu. Les rôles doivent inclure au moins un de ces répertoires, mais il est parfaitement fonctionnel d'exclure ceux qui ne sont pas nécessaires. Lorsqu'il est utilisé, chaque répertoire doit contenir un fichier `main.yml`, qui contient le contenu adéquat :

- `tasks` - contient la liste principale des tâches à exécuter par le rôle.
- `handlers` - contient les `handlers` pouvant être utilisés par ce rôle ou même en dehors de ce rôle.
- `defaults` - variables par défaut du rôle (basse priorité)
- `vars` - autres variables pour le rôle.
- `files` - contient des fichiers pouvant être déployés via ce rôle.
- `templates` - contient des modèles pouvant être déployés via ce rôle.
- `meta` - définit des métadonnées pour ce rôle, notamment les dépendances.

D’autres fichiers YAML peuvent être inclus dans certains répertoires. Par exemple, il est courant d’inclure des tâches spécifiques à la plate-forme à partir du fichier `tasks/main.yml` :

```
# roles/example/tasks/main.yml
- name: added in 2.4, previously you used 'include'
  import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'
- import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'

# roles/example/tasks/redhat.yml
- yum:
    name: "httpd"
    state: present

# roles/example/tasks/debian.yml
- apt:
    name: "apache2"
    state: present
```

On appelle des rôles à partir d’un livre de jeu par exemple `webservers.yml` :

```
---
# file: webservers.yml
- hosts: webservers
  roles:
    - common
    - webtier
```

Le livre de jeu principal de l’infrastrucure `site.yml` serait le suivant :

```
---
# file: site.yml
- import_playbook: webservers.yml
- import_playbook: dbservers.yml
```

3.6. include_role

Le module include_role permet de charger un rôle comme une tâche dynamique.

```
- name: Run tasks/other.yaml instead of 'main'
  include_role:
    name: myrole
    tasks_from: other
```

3.7. Création d’un rôle

Création d’un rôle :

```
cd roles
ansible-galaxy init newrole
- newrole was created successfully
```

```
tree newrole/
newrole/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml

8 directories, 8 files
```

4. Tags Ansible

Tags

4.1. Tags Ansible

Si vous avez un grand livre de jeu, il peut s'avérer utile de ne pouvoir en exécuter qu'une partie spécifique plutôt que de tout lire dans le livre. Ansible prend en charge un attribut `tags:` pour cette raison.

Lorsque vous exécutez un livre de jeu, vous pouvez filtrer les tâches en fonction des "tags" de deux manières:

- Sur la ligne de commande, avec les options `--tags` ou `--skip-tags`
- Dans les paramètres de configuration Ansible, avec les options `TAGS_RUN` et `TAGS_SKIP`

Les "tags" peuvent être appliqués à de nombreuses structures dans Ansible :

- `tasks:`
- `roles:`
- `blocks:`
- `import_roles:`
- `import_tasks:`

Mais son utilisation la plus simple est avec des tâches individuelles. Voici un exemple qui balise deux tâches avec des "tags" différentes:

```
- hosts: host
  tasks:
    - yum:
        name:
          - httpd
          - memcached
        state: installed
      tags:
       - packages

    - template:
        src: templates/src.j2
        dest: /etc/foo.conf
      tags:
```

```
    - configuration
```

Si vous souhaitez simplement exécuter les parties “configuration” et “packages” d’un très long livre de jeu, vous pouvez utiliser l’option `--tags` sur la ligne de commande :

```
ansible-playbook example.yml --tags "configuration,packages"
```

Si vous souhaitez exécuter un livre de jeu sans certaines tâches marquées, vous pouvez utiliser l’option de ligne de commande `--skip-tags` :

```
ansible-playbook example.yml --skip-tags "packages"
```

4.2. Réutilisation des tags

Vous pouvez appliquer le même “tag” à plusieurs tâches. Lors de l’exécution d’une livre de jeu à l’aide de l’option de ligne de commande `--tags`, toutes les tâches portant ce nom de “tag” seront exécutées. Cet exemple balise plusieurs tâches avec un “tag” “ntp” :

```
---
# file: roles/common/tasks/main.yml

- name: be sure ntp is installed
  yum:
    name: ntp
    state: installed
  tags: ntp

- name: be sure ntp is configured
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
  notify:
    - restart ntpd
  tags: ntp

- name: be sure ntpd is running and enabled
  service:
    name: ntpd
```

```
    state: started
    enabled: yes
  tags: ntp
```

4.3. Héritage des tags

On peut ajouter des “tags” à un jeu ou à des rôles ou des tâches importées statiquement. Toutes leurs tâches hériterons des ces “tags”.

L’héritage des “tags” ne s’applique pas aux inclusions dynamiques comme `include_role` et `include_tasks`.

On peut aussi appliquer des tags aux autres structures que les tâches mais retenons que ces “tags” sont fondamentalement appliqués aux tâches.

Cet exemple identifie toutes les tâches des deux jeux. La première partie dispose de toutes les tâches étiquetées avec “bar”, et la seconde toutes les tâches étiquetées avec “foo” :

```
- hosts: all
  tags:
    - bar
  tasks:
    ...

- hosts: all
  tags: ['foo']
  tasks:
    ...
```

On peut aussi appliquer les “tags” aux tâches importées par des rôles :

```
roles:
  - role: webserver
    vars:
      port: 5000
    tags: [ 'web', 'foo' ]
```

Ainsi que sur les directives `import_role:` et `import_tasks:` :

```
- import_role:
    name: myrole
  tags: [web,foo]

- import_tasks: foo.yml
  tags: [web,foo]
```

Ces “tags” sont appliqués aux tâches par héritage. Cela sert à exécuter un livre de jeu de manière sélective.

Les “tags” sont appliqués tout au long de la chaîne de dépendance. Pour qu’un “tag” soit hérité des tâches d’un rôle dépendant, il doit être appliqué à la déclaration de rôle ou à l’importation statique, et non à toutes les tâches du rôle.

Vous pouvez voir quelles “tags” sont appliqués aux tâches, aux rôles et aux importations statiques en exécutant `ansible-playbook` avec l’option `--list-tasks`. Vous pouvez afficher toutes les balises appliquées aux tâches avec l’option `--list-tags`.

L’héritage ne fonctionnant pas avec `include_tasks`, `include_roles` et les autres “includes” dynamiques, ceux-ci doivent être appliqués sur chaque tâche ou sur des `blocks:`

Voici un exemple de marquage de tâches de rôle avec le “tag” “mytag”, à l’aide d’une instruction de bloc, à utiliser ensuite avec un “include” dynamique:

```
- hosts: all
  tasks:
  - include_role:
      name: myrole
    tags: mytag
```

```
- block:
    - name: First task to run
    ...
    - name: Second task to run
    ...
  tags:
    - mytag
```

4.4. Tags spéciaux

- “always” : toujours exécuté tant qu’il n’est explicitement évité (`--skip-tags always`)
- “never” : jamais exécuté tant qu’il n’est explicitement exécuté (`--tags never`)
- “tagged” : exécute uniquement les tâches balisées
- “untagged” : exécute les tâches qui ne sont pas balisées uniquement.
- “all” : exécute toutes les tâches balisées ou non

Note : Ansible fonctionne par défaut comme si `--tags all` était précisé.

5. Ansible Vault

Ansible-Vault et Variables and Vaults best practices sont des références à visiter.

5.1. Présentation de la problématique

Ansible-vault est un outil intégré à Ansible qui permet de chiffrer les fichiers qui contiennent des données sensibles.

Il y a certainement beaucoup d’approches pour protéger des variables confidentielles utilisées dans les livres de jeu :

- Selon un certain niveau de sécurité, on pourra utiliser des variables d’environnement (notamment pour stocker des TOKEN).
- Chiffrer un fichier de variables secrètes et encoder un mot de passe à chaque usage.
- Chiffrer la valeur uniquement.
- …

Une autre approche fonctionnelle consisterait à placer son mot de passe dans un fichier dans un emplacement protégé avec des droits restreints, en tout cas en dehors du livre de jeu et de dépôt de contrôle de version. Il s’agira d’appeler ce fichier à chaque exécution du livre de jeu. Un fichier de variables confidentielle aura été chiffré avec `ansible-vault` à l’aide du fichier qui contient le mot de passe. Ce fichier de variables est inclus comme d’habitude dans un jeu ou ailleurs comme tout fichier de variables.

Une approche selon nous plus complexe est évoquée ici comme “Best Practice” : Variables and Vaults best practices

5.2. Approche par fichiers de mot de passe et de variables confidentiels

Placer ses variables confidentielles dans un fichier (ici `secret.yml`) :

```
secret: |
  Ceci est le message chiffré sur plusieurs lignes
  Voici la seconde ligne
```

Créer un fichier qui contient les mots de passe :

```
echo "testtest" > ~/.vault_passwords.txt ; chmod 600 ~/.vault_passwords.txt
```

Chiffrer le fichier `secret.yml` avec le mot de passe contenu dans l'endroit protégé `~/.vault_passwords.txt` :

```
ansible-vault encrypt secret.yml --vault-password-file ~/.vault_passwords.txt
```

Pour preuve :

```
cat secret.yml | head -n 3
$ANSIBLE_VAULT;1.1;AES256
38633337663964663931613733393934383236653434393534393035643137333966633061653762
35363264656439303430626630653035383938313538633 60a303036396362353934353163633730
```

Playbook de démonstration :

```
---
- name: "demo ansible-vault"
  hosts: localhost
  vars_files:
    - secret.yml
  gather_facts: False
  tasks:
    - name: "affiche le secret"
      debug:
        msg: "{{ secret }}"
```

Le message chiffré s'affiche !

```
ansible-playbook test-vault.yml --vault-password-file ~/.vault_passwords.txt

PLAY [demo ansible-vault] ******************************************************

TASK [affiche le secret] *******************************************************
ok: [localhost] => {
    "msg": "Ceci est le message chiffré sur plusieurs lignes\nVoici la seconde ligne\n"
}

PLAY RECAP *********************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0
```

5.3. Fichier de mot de passe par défaut

On peut indiquer dans la configuration de Ansible l'emplacement par défaut du fichier de mot de passe via la directive `vault_password_file = /path/to/vault_password_file`.

5.4. Garder les variables protégées visibles en toute sécurité

On devrait chiffrer les variables sensibles ou secrètes avec Ansible Vault. Mais en chiffrant les noms des variables ainsi que leurs valeurs, il est difficile de trouver la source de ces valeurs. On pourrait garder les noms de vos variables accessibles (par grep, par exemple) sans révéler de secrets en ajoutant une couche indirecte[1] :

- Créer un sous-répertoire dans `group_vars/` portant le nom du groupe.
- Dans ce sous-répertoire, créer deux fichiers nommés `vars` et `vault`.
- Dans le fichier `vars`, définir toutes les variables nécessaires, y compris les variables sensibles.
- Copier toutes les variables sensibles dans le fichier `vault` et préfixer ces variables avec `vault_`.
- Ajuster les variables dans le fichier `vars` pour qu'elles pointent vers les variables `vault_` correspondantes en utilisant la syntaxe Jinja2 : `db_password: {{ vault_db_password }}`.
- Chiffrer le fichier `vault` pour protéger son contenu.
- Utiliser le nom de variable du fichier vars dans le livre de jeu.

5.5. Articles

- Hashicorp Vault & Ansible https://github.com/jhaals/ansible-vault
- `ansible-vault` workflow described https://gist.github.com/tristanfisher/e5a306144a637dc739e7

1. Keep vaulted variables safely visible ↩

Revision #1
Created 3 October 2021 21:12:53 by garfi
Updated 3 October 2021 21:13:53 by garfi