Create encrypted backups of your workstation with an Ansible playbook

Jose Vicente Nunez ⋮ 27-34 minutes ⋮ 2/10/2023

Posted: February 10, 2023

I like to make backups. No, there's nothing wrong with me. Having a failsafe before doing something that can't be rolled back easily, like installing a new version of your favorite operating system, should be compulsory.

Also, I don't like to repeat myself. For that reason, I like to write scripts so that a successful experience can be repeated. Here is one:

```
#!/bin/bash
set -e # exit on errors

# Partition the disk
sudo fdisk /dev/sda
sudo partprobe /dev/sda

# Encrypt the partition
sudo cryptsetup luksFormat /dev/sda1
sudo cryptsetup luksOpen /dev/sda1 usbluks

# Create a logical volume
sudo vgcreate usbluks_vg /dev/mapper/usbluks
sudo lvcreate -n usbluks_logvol -L 118G+ usbluks_vg

# Format the disk with btrfs
sudo mkfs.btrfs --mixed \
--data SINGLE --metadata SINGLE \
/dev/usbluks_vg/usbluks_logvol

# Mount our new volume
sudo mount /dev/usbluks_vg/usbluks_logvol /mnt/

# Backup my files
sudo tar --create --directory \
/home --file -| tar --directory \
/mnt --extract --file -
```

Not bad, with bare-minimum error handling. Bash was my first choice because it is simple yet expressive. But Bash stops being the ideal choice for this task when you compare it to another tool that I like, Ansible.

Ansible's advantages include:

1. It provides more complete debugging. I'm not just talking about `printf` statements but also richer error handling tools with a detailed stack trace.
2. It offers idempotence.
3. It can group your operations in blocks.
4. It offers control over how your script runs: tags, limit where it runs, do a dry run.
5. It has code-smell-detection tools. Bash has great tools out there, but Ansible has ansible-lint and check flag.
6. It provides remote execution. While my script doesn't need it, if I want to run this on a different group of machines, I only need to make a few changes. Ansible's support for remote task execution is unmatched.

I began to rethink my script. Ansible's capabilities made me realize that improving my Bash version would probably be too verbose and more difficult to maintain than one written in Ansible. So I set out to do that.

Before I get started, though, allow me to talk a little about pet versus cattle servers.

[Write your first playbook in this hands-on interactive lab.]

What's the difference between pet and cattle servers?

There is an ongoing debate about how you should treat your servers (as either pets or cattle). There are many good analogies and explanations out there (including Randy Bias' definitions, which I quote below).

According to Randy, pet servers are:

> Servers or server pairs that are treated as indispensable or unique systems that can never be down. Typically they are manually built, managed, and "hand fed." Examples include mainframes, solitary servers, [high availaiblity] loadbalancers/firewalls (active/active or active/passive), database systems designed as ... active/passive, and so on.

Bash and Python are perfect tools to automate tasks on pet servers, as they are very expressive and you are not very concerned about uniformity. This doesn't mean you cannot have an Ansible playbook for such specialized servers; it may be an excellent reason to have a dedicated playbook for them.

Cattle servers, Randy writes, are:

> Arrays of more than two servers that are built using automated tools and are designed for failure, where no one, two, or even three servers are irreplaceable. Typically, during failure events no human intervention is required as the array exhibits attributes of "routing around failures" by restarting failed servers or replicating data through strategies like triple replication or erasure coding. Examples include web server arrays, multi-master datastores such as Cassandra clusters, multiple racks of gear put together in clusters, and just about anything that is load-balanced and multi-master.

Ansible and other specialized provisioning tools ensure that the state of the servers is uniform and doesn't change by accident.

Back up your workstation with Ansible

So, a home computer is probably a pet server. And you might be wondering, is Ansible a bad tool for handling pet servers? No, it is actually quite good.

This article shows how you can also use an Ansible playbook to make an encrypted backup of your home directory to keep your workstation properly backed up. It borrows heavily from an article by Peter Gervaise, so spend some time reading it.

Requirements

- sudo (to run programs that require elevated privileges, such as creating a filesystem or mounting a disk)
- Python 3 and pip
- Ansible 2
- A USB drive (it doesn't matter if it comes formatted; this process will remove all the files on it)
- Curiosity

I will show a few changes you need to make, as Ansible is normally suited for cattle servers. For pets, you need to make a few changes to the playbook.

[Want to test your sysadmin skills? Take a skills assessment today.]

Install Ansible

If you don't have Ansible installed, you can do something like this:

```
$ python3 -m venv ~/virtualenv/ansible
. ~/virtualenv/ansible/bin/activate
```

```
$ python -m pip install --upgrade pip

$ pip install -r requirements.txt
```

And if the **community.general** Ansible Galaxy module is not present:

```
$ ansible-galaxy collection list 2>&1|rg general # Comes empty

$ ansible-galaxy collection install community.general
```

Explore the Ansible playbook

You have probably seen Ansible playbooks before, but this one has a few special features:

1. It doesn't use Secure Shell (SSH) to connect to a remote host. The "remote" host is **localhost** (the same machine where the playbook runs), so you use a special kind of connection called **local**.
2. It uses special fact-gathering to speed up the device's feature detection. I will elaborate more in the next section.
3. You can define variables to make the playbook reusable, but you need to prompt the user for the values instead of defining them on the command line. Pet servers have unique features, so it's better to ask the user for some choices interactively as the playbook runs.
4. Use tags. If you just want to run parts of this playbook, you can skip to the desired target (`ansible-playbook --tag $mytag encrypted_us_backup.yaml`).

Get your facts straight

Every time you run an Ansible playbook, it collects facts about your target system to operate properly. That is overkill for this task; you need information only about the devices, and you can ignore other details like DNS or Python.

First, disable the general **gather_facts**, then override it with the task below using the **setup** module to enable devices and mounts.

After that, you can use the facts in any way you see fit (check `fact_filtering.yaml`) to see how to do it):

```
---
- name: Restricted fact gathering example
  hosts: localhost
  connection: local
  become: true
  gather_facts: false
  vars_prompt:
    - name: device
      prompt: "Enter name of the USB device"
      private: false
      default: "sda"
  vars:
    target_device: "/dev/{{ device }}"
  tasks:
    - name: Get only 'devices, mount' facts
      ansible.builtin.setup:
        gather_subset:
          - '!all'
          - '!min'
```

```
          - devices
          - mounts
    - name: Basic setup and verification for target system
      block:
        - name: Facts for {{ target_device }}
          community.general.parted:
            device: "{{ target_device }}"
            state: "info"
            unit: "GB"
          register: target_parted_data
        - name: Calculate disk size
          debug:
            msg: "{{ ansible_devices[device] }}"
        - name: Calculate available space on USB device and save it as a fact
          ansible.builtin.set_fact:
            total_usb_disk_space: "{{ (ansible_devices[device]['sectorsize']|
int) * (ansible_devices[device]['sectors']|int) }}"
            cacheable: yes
          when: target_parted_data is defined
        - name: Print facts for {{ target_device }}
          ansible.builtin.debug:
            msg: "{{ ansible_devices[device].size }}, {{ total_usb_disk_space }}
bytes"
          when: target_parted_data is defined
```

See it in action:

Now that you know the disk size, you should also calculate how much disk space the backup will consume. Ansible doesn't have a du task, so wrap your own.

[Learn about upcoming webinars, in-person events, and more opportunities to increase your knowledge at Red Hat events.]

How much disk space do you need?

To calculate your storage needs for the backup, you need a couple of things:

1. Capture the output of the du command (see the `disk_usage_dir.yaml` file), and report if it changes only the return code.
2. Filter the output of the du command, so you can use it later.

```
---
- name: Disk utilization capture
  hosts: localhost
  connection: local
  become: true
  gather_facts: false
  vars_prompt:
    - name: source_dir
      prompt: "Enter name of the directory to back up"
      private: false
      default: "/home/josevnz"
  tasks:
    - name: Capture disk utilization on {{ source_dir }}
      block:
```

```
        - name: Get disk utilization from {{ source_dir }}
          ansible.builtin.command:
            argv:
              - /bin/du
              - --max-depth=0
              - --block-size=1
              - --exclude='*/.cache'
              - --exclude='*/.gradle/caches'
              - --exclude='*/Downloads'
              - "{{ source_dir }}"
          register: du_capture
          changed_when: "du_capture.rc != 0"
        - name: Process DU output
          ansible.builtin.set_fact:
            du: "{{ du_capture.stdout | regex_replace('\\D+') | int }}"
        - name: Print facts for {{ target_device }}
          ansible.builtin.debug:
            msg: "{{ source_dir }} -> {{ du }} bytes"
          when: du_capture is defined
```

Here it is running:

With this information, you can verify whether the USB drive has enough space to save the files:

```
        - name: Check if destination USB drive has enough space to store our
backup
          ansible.builtin.assert:
            that:
              - ( total_usb_disk_space | int ) > ( du | int)
            fail_msg: "Not enough disk space on USB drive! {{ du | int |
human_readable() }} > {{ total_usb_disk_space | int | human_readable() }}"
            success_msg: "We have enough space to make the backup!"
          tags: disk_space_check
```

If the destination is too small, it may cancel the whole operation:

So you are good to go, right? We live in an imperfect world where counterfeit USB drives are sold. Manufacturers do tricks to advertise higher capacity than is really supported.

Using the open source tool f3, you can run this with brand-new media to ensure the capacity is indeed what you think you purchased. I will cover that next.

Trust but verify (that the disk is legitimate)

Putting it in another way:

> To test men and verify what has actually been done—this, this again this alone is now the main feature of all our activities, of our whole policy. —paraphrasing Vladimir Lenin

Depending on the size of the disk, this task can be quick or take a long time:

```
  tasks:
    - name: Verify with f3 {{ device }}
      block:
        - name: Testing with f3 {{ device }}
```

```
          ansible.builtin.command:
            argv:
              - /usr/bin/f3probe
              - "{{ target_device }}"
          register: f3_capture
          changed_when: "f3_capture.rc != 0"
          when: paranoid is defined and paranoid == "y"
        - name: Print facts for {{ target_device }}
          ansible.builtin.debug:
            msg: "{{ target_device }} -> {{ f3_capture }}"
          when: f3_capture is defined
```

I don't want to force this task every time I run my encrypted backup with media I know is good, so unless I enable it in the prompt, it will not run.

Here is the `running_f3.yaml` file:

```
$ ansible-playbook running_f3.yaml
Enter name of the USB device [sda]:
Check the USB drive real capacity with f3 (y/n) [n]: y

PLAY [USB drive verification] ***************************

TASK [Testing with f3 sda] ****************************
ok: [localhost]

TASK [Print facts for /dev/sda] ***************************
ok: [localhost] => {

"msg": "/dev/sda -> {'changed': False, 'stdout': 'F3 probe 8.0\\nCopyright (C)
2010 Digirati Internet LTDA.\\nThis is free software; see the source for copying
conditions.\\n\\nWARNING: Probing normally takes from a few seconds to 15
minutes, but\\n         it can take longer. Please be patient.\\n\\nProbe
finished, recovering blocks... Done\\n\\nGood news: The device `/dev/sda\\' is
the real thing\\n\\nDevice geometry:\\n\\t [...]

*Usable* size: 960.00 MB (1966080 blocks)\\n\\t
Announced size: 960.00 MB (1966080 blocks)\\n\\t
[...]
}

PLAY RECAP ****************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

This step took over five minutes on a small but slow USB drive.

Put everything together and write the final playbook

Here is how you can implement these requirements.

First, capture user choices interactively:

```
---
- name: Take an empty USB disk and copy user home directory into an encrypted
partition
  hosts: localhost
  connection: local
  become: true
  gather_facts: false
  any_errors_fatal: true
  vars_prompt:
    - name: source_dir
      prompt: "Enter name of the directory to back up"
      private: false
      default: "/home/josevnz"
    - name: device
      prompt: "Enter name of the USB device"
      private: false
      default: "sda"
    - name: passphrase
      prompt: "Enter the passphrase to be used to protect the target device"
      private: true
    - name: destroy_partition
      prompt: "Destroy any existing partitions (y/n)?"
      default: "y"
      private: false
    - name: cryptns
      prompt: "Name of the luks device. Strongly suggested you pick one"
      default: "{{ query('community.general.random_string', upper=false,
numbers=false, special=false, length=4)[0] }}"
      private: false
    - name: paranoid
      prompt: "Check the USB drive real capacity with f3 (y/n)"
      default: "n"
      private: false
  vars:
    volgrp: "{{ cryptns }}_vg"
    logicalvol: "backuplv"
    cryptns: "usbluks"
    cryptdevice: "/dev/mapper/{{ cryptns }}"
    destination_dir: "/mnt"
    target_device: "/dev/{{ device }}"
```

The next step is to define the more actions. Get partition details, validate devices, and destroy the destination if it already exists. I use a block to group these operations (like installing cryptsetup at the end, if it is missing):

```
  tasks:
    - name: Basic setup and verification for target system
      block:
        - name: Get only 'devices, mount' facts
          ansible.builtin.setup:
            gather_subset:
              - '!all'
              - '!min'
```

```
            - devices
            - mounts
          tags: facts_subset
        - name: Facts for {{ target_device }}
          community.general.parted:
            device: "{{ target_device }}"
            state: "info"
            unit: "GB"
          register: target_parted_data
          tags: parted_data
        - name: Calculate available space on USB device and save it as a fact
          ansible.builtin.set_fact:
            total_usb_disk_space: "{{ (ansible_devices[device]['sectorsize'] |
int) * (ansible_devices[device]['sectors'] | int) }}"
            cacheable: true
          when: target_parted_data is defined
          tags: space_usb
        - name: Get disk utilization from {{ source_dir }}
          ansible.builtin.command:
            argv:
              - /bin/du
              - --max-depth=0
              - --block-size=1
              - --exclude='*/.cache'
              - --exclude='*/.gradle/caches'
              - --exclude='*/Downloads'
              - "{{ source_dir }}"
          register: du_capture
          changed_when: "du_capture.rc != 0"
          tags: du_capture
        - name: Process DU output
          ansible.builtin.set_fact:
            du: "{{ du_capture.stdout | regex_replace('\\D+') | int }}"
            type: integer
          tags: du_filter
        - name: Check if destination USB drive has enough space to store our
backup
          ansible.builtin.assert:
            that:
              - ( total_usb_disk_space | int ) > ( du | int)
            fail_msg: "Not enough disk space on USB drive! {{ du | int |
human_readable() }} > {{ total_usb_disk_space | int | human_readable() }}"
            success_msg: "We have enough space to make the backup!"
          tags: disk_space_check
        - name: Facts for {{ target_device }}
          community.general.parted:  # Note this task doesn't use facts
            device: "{{ target_device }}"
            state: "info"
            unit: "GB"
          register: target_parted_data
          tags: parted_info
        - name: Print facts for {{ target_device }}
          ansible.builtin.debug:
            msg: "{{ target_parted_data }}"
```

```
          when: target_parted_data is defined
          tags: print_facts
        - name: Unmount USB drive
          ansible.posix.mount:
            path: "{{ destination_dir }}"
            state: absent
          when: target_parted_data is defined
          tags: unmount
        - name: Install f3 (test for fake flash drives and cards)
          ansible.builtin.dnf:
            name: f3
            state: installed
          tags: f3install
        - name: Check if USB device is a fake one with f3
          ansible.builtin.command: "/usr/bin/f3probe {{ target_device }}"
          register: f3_run
          when: target_parted_data is defined and paranoid is defined and
paranoid == "y"
          tags: f3_fake
        - name: Destroy existing partition if found, update target_parted_data
          community.general.parted:
            device: "{{ target_device }}"
            number: 1
            state: "absent"
            unit: "GB"
          when: target_parted_data is defined and destroy_partition == "y"
          register: target_parted_data
          tags: destroy_partition
        - name: "Abort with an error If there are still any partitions for {{
target_device }}"
          ansible.builtin.debug:
            msg: -|
              {{ target_device }} is already partitioned, {{
target_parted_data['partitions'] }}.
              Size is {{ target_parted_data['disk']['size'] }} GB
          failed_when: target_parted_data['partitions'] is defined and
(target_parted_data['partitions'] | length > 0) and not ansible_check_mode
          tags: fail_on_existing_part
    - name: Get destination device details
```

Create the partition and print a little information:

```
    - name: Create partition
      block:
        - name: Get info on destination partition
          community.general.parted:
            device: "{{ target_device }}"
            number: 1
            state: info
          register: info_output
          tags: parted_info
        - name: Print info_output
          ansible.builtin.debug:
```

```
            msg: "{{ info_output }}"
          tags: parted_print
        - name: Create new partition
          community.general.parted:
            device: "{{ target_device }}"
            number: 1
            state: present
            part_end: "100%"
          register: parted_output
          tags: parted_create
      rescue:
        - name: Parted failed
          ansible.builtin.fail:
            msg: 'Parted failed:  {{ parted_output }}'
```

Next, create the encrypted volume, format it, and mount it.

Another block:

```
    - name: LUKS and filesystem tasks
      block:
        - name: Install crysetup
          ansible.builtin.dnf:
            name: cryptsetup
            state: installed
          tags: crysetup
        - name: Create LUKS container with passphrase
          community.crypto.luks_device:
            device: "{{ target_device }}1"
            state: present
            name: "{{ cryptns }}"
            passphrase: "{{ passphrase }}"
          tags: luks_create
        - name: Open luks container
          community.crypto.luks_device:
            device: "{{ target_device }}1"
            state: opened
            name: "{{ cryptns }}"
            passphrase: "{{ passphrase }}"
          tags: luks_open
        - name: Create {{ cryptdevice }}
          community.general.system.lvg:
            vg: "{{ volgrp }}"
            pvs: "{{ cryptdevice }}"
            force: true
            state: "present"
          when: not ansible_check_mode
          tags: lvm_volgrp
        - name: Create a logvol in my new vg
          community.general.system.lvol:
            vg: "{{ volgrp }}"
            lv: "{{ logicalvol }}"
            size: "+100%FREE"
```

```
          when: not ansible_check_mode
          tags: lvm_logvol
        - name: Create a filesystem for the USB drive (man mkfs.btrfs for
options explanation)
          community.general.system.filesystem:
            fstype: btrfs
            dev: "/dev/mapper/{{ volgrp }}-{{ logicalvol }}"
            force: true
            opts: --data SINGLE --metadata SINGLE --mixed --label backup --
features skinny-metadata,no-holes
          when: not ansible_check_mode
          tags: mkfs
        - name: Mount USB drive (use a dummy fstab to avoid changing real
/etc/fstab)
          ansible.posix.mount:
            path: "{{ destination_dir }}"
            src: "/dev/mapper/{{ volgrp }}-{{ logicalvol }}"
            state: mounted
            fstype: btrfs
            fstab: /tmp/tmp.fstab
          when: not ansible_check_mode
          tags: mount
```

You do not want to persist the status of the mounted filesystem across reboots; you want to unmount the drive as soon it's done with the backup. For this reason, use a temporary fstab file (`fstab: /tmp/tmp.fstab`).

The final task is to create the backup on the new encrypted volume using synchronize, a frontend for `rsync`.

[Related reading: Keeping Linux files and directories in sync with rsync]

Pass a few arguments to skip unwanted directories:

```
    - name: Backup stage
      tags: backup
      block:
        - name: Backup using rsync
          ansible.posix.synchronize:
            archive: true
            compress: false
            dest: "{{ destination_dir }}"
            owner: true
            partial: true
            recursive: true
            src: "{{ source_dir }}"
            rsync_opts:
              - "--exclude Downloads"
              - "--exclude .cache"
              - "--exclude .gradle/caches"
```

Take this for a dry run before running it on your USB drive.

Fix style issues and mistakes, take it for a test drive

You can test this playbook before running it.

First, see if `ansible-lint` has any complaints:

```
$ ~/virtualenv/EncryptedUsbDriveBackup/bin/ansible-lint
encrypted_usb_backup.yaml
WARNING  Overriding detected file kind 'yaml' with 'playbook' for given
positional argument: encrypted_usb_backup.yaml

Passed with production profile: 0 failure(s), 0 warning(s) on 1 files.
```

Then do a dry run:

```
$ ansible-playbook --check encrypted_usb_backup.yaml
```

Make the backup and do a full run

Below is an example of what your backup session may look like (without the f3 and backup verification):

```
$ ansible-playbook encrypted_usb_backup.yaml
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'
Enter name of the directory to back up [/home/josevnz]:
/home/josevnz/Documents/Domino/
Enter name of the USB device [sda]: sdc
Enter the passphrase to be used to protect the target device:
Destroy any existing partitions (y/n)? [y]:
Name of the luks device. Strongly suggested you pick one [rfjz]:
Check the USB drive real capacity with f3 (y/n) [n]:

PLAY [Take an empty USB disk and copy user home directory into an encrypted
partition] ***************************************

TASK [Get only 'devices, mount' facts] *********************************
ok: [localhost]

TASK [Facts for /dev/sdc] **********************************
ok: [localhost]

TASK [Calculate available space on USB device and save it as a fact]
**********************************
ok: [localhost]

TASK [Get disk utilization from /home/josevnz/Documents/Domino/]
**********************************
ok: [localhost]

TASK [Process DU output]
**********************************
ok: [localhost]

TASK [Check if destination USB drive has enough space to store our backup]
**********************************
```

```
ok: [localhost] => {
    "changed": false,
    "msg": "We have enough space to make the backup!"
}

TASK [Facts for /dev/sdc]
*********************************
ok: [localhost]

TASK [Print facts for /dev/sdc]
*********************************
ok: [localhost] => {
    "msg": {
        "changed": false,
        "disk": {
            "dev": "/dev/sdc",
            "logical_block": 512,
            "model": "General UDisk",
            "physical_block": 512,
            "size": 1.01,
            "table": "msdos",
            "unit": "gb"
        },
        "failed": false,
        "partitions": [
            {
                "begin": 0.0,
                "end": 1.01,
                "flags": [],
                "fstype": "",
                "name": "",
                "num": 1,
                "size": 1.01,
                "unit": "gb"
            }
        ],
        "script": "unit 'GB' print"
    }
}

TASK [Unmount USB drive]
*********************************
ok: [localhost]

TASK [Install f3 (test for fake flash drives and cards)]
*********************************
ok: [localhost]

TASK [Check if USB device is a fake one with f3]
*********************************
skipping: [localhost]

TASK [Destroy existing partition if found, update target_parted_data]
```

```
*********************************
changed: [localhost]

TASK [Abort with an error If there are still any partitions for /dev/sdc]
*********************************
ok: [localhost] => {
    "msg": "-| /dev/sdc is already partitioned, []. Size is 1.01 GB"
}

TASK [Get info on destination partition] *********************************
ok: [localhost]

TASK [Print info_output] *********************************
ok: [localhost] => {
    "msg": {
        "changed": false,
        "disk": {
            "dev": "/dev/sdc",
            "logical_block": 512,
            "model": "General UDisk",
            "physical_block": 512,
            "size": 983040.0,
            "table": "msdos",
            "unit": "kib"
        },
        "failed": false,
        "partitions": [],
        "script": "unit 'KiB' print"
    }
}

TASK [Create new partition] *********************************
changed: [localhost]

TASK [Install crysetup] *********************************
ok: [localhost]

TASK [Create LUKS container with passphrase] *********************************
ok: [localhost]

TASK [Open luks container] *********************************
changed: [localhost]

TASK [Create /dev/mapper/rfjz] *********************************
changed: [localhost]

TASK [Create a logvol in my new vg] *********************************
changed: [localhost]

TASK [Create a filesystem for the USB drive (man mkfs.btrfs for options
explanation)] *********************************
changed: [localhost]

TASK [Mount USB drive (use a dummy fstab to avoid changing real /etc/fstab)]
```

```
*********************************
changed: [localhost]

TASK [Backup using rsync] *********************************
changed: [localhost]

TASK [Facts for /dev/sdc] *********************************
skipping: [localhost]

PLAY RECAP *********************************
localhost                  : ok=19   changed=8    unreachable=0    failed=0
skipped=3    rescued=0    ignored=0
```

Next steps

Where can you go from here? Here are a few ideas:

- If you want to use a graphical user interface (GUI) for your encrypted backup, consider VeraCrypt. It is open source and has a nice wizard to walk you through the process. There is a tutorial on how to do that.
- Instead of local storage, use the cloud (but encrypt first). I used `rsync` to back up to a USB drive, but Ansible can also upload files to an S3 cloud volume. Ideally, you should make an archive and encrypt it with sops before the upload.
- If you want to back up more than one user directory, you could use a loop.
- You can check whether the USB has bad blocks before or after making the backup. I wrote a small Ansible playbook that you can run to see how the program bad blocks works (see `verify_usb.yaml`).
- The source code for the complete Ansible playbook (`encrypted_usb_backup.yaml`) is in my GitHub repo for this project. Feel free to download and improve it for your use case.

[Get the YAML cheat sheet]