Salt and Napalm based Network Device Management

February 2019
written by Claudius Zingerli

I recently got some time to spend looking at network automation using salt a napalm. We're using SaltStack's salt for automation and config management for more than two years now and are traditionally a network engineering company, so why not combine the two areas and automate tasks in network engineering?

Traditionally in enterprises as well as ISPs, there are high numbers of network devices of multiple generations and from multiple vendors. An admin's nightmare to maintain manually. So: An ideal target for automation. But automation is about getting rid of repetitive tasks. As vendors have evolved, different ways to perform tasks were found. This makes automation challenging here. Luckily there is NAPALM – an abstraction layer that provides a generic API and uses vendor specific drivers to interact with the network devices. This article provides a tutorial on integrating salt and NAPALM applied to an easy example of maintaining user accounts and credentials on the devices.

Story

I'm a bad storyteller, but if you like:

- We had to perform some changes on a customers network that was only partly documented. In order to be able to create a link graph, we would like to use LLDP. LLDP was enabled on some newly installed devices, but still missing on many. Devices without LLDP support typically just pass through LLDP traffic of connected devices – and therefore won't appear in the graph and strange topologies will result (ever seen WLAN access points connected to core routers? Yes, hundreds!). NAPALM provides means to gather LLDP information – together with salt – nicely formatted as JSON to be used fed into hip tools.
- There is another story that we will base our tutorial on: We operate a few lab switches in our

office to let our engineers perform tests experiment with technology. We want to unify authentication configuration and simplify config restore in case somebody kills a device.

(yes, both stories are true)

Motivation

Why are we using salt

- It is a config management and an orchestration solution: It can push configuration changes but also react to state changes
- We use it in other projects; it is the new cool kid (since 2016 or so?)
- It scales well (it has standing connections up- and downstream; most of the work is performed client side)
- Some fuel the automation wars: It can be agent-based, agent-less, proxy, ssh, hierarchical, $whatever you need.
- It has a very active community and a company backing it, it is easily extensible and offers a large existing plugin base.

Why NAPALM

- Salt provides dedicated proxy modules for Arista, Junos, Cisco and several other network OSs, each with its own syntax and semantics.
- We want to use the same syntax and features, so we chose the additional abstraction layer 'napalm'.
- It is under active development, mostly by original author Mircea Ulinic of Cloudflare.

Why salt-napalm

- It requires only a single point of entry to the infrastructure. This simplifies ACLs (egress TCP from the proxy-host to the salt-master)
- Integration with the existing server management solution (One tool for the task, reuse existing infrastructure (backup, monitoring, test environment, agents)
- Deepen knowledge of employees; bring together our Network Engineers and System Admins.

Tutorial

Setup

Before starting, we need to make sure some preconditions are met:

- There is a basic understanding of salt, its architecture and terminology. If not, please refer to the extensive documentation.
- There is already a salt-master running with default config (States in /srv/salt, Pillar in /srv/pillar) and has the napalm-install-formula integrated.
- There is a salt-minon running on proxy-host-* and connected to the salt-master.
- There is at least one network device swi-* that you can ssh into from the host running salt-minion.

We are using salt 2018.3.3, napalm 2.4.0 on Debian 9 and network devices running Junos 15.1 and 17.3. The proxy-host is an APU2 of PCengines.

Configuration on the Salt Master

Salt-napalm provides a formula that simplifies napalm setup on the proxy-host:
https://github.com/saltstack-formulas/napalm-install-formula

We use this formula to setup the tools.

Configuration on the Salt Master

Salt-napalm provides a formula that simplifies napalm setup on the proxy-host:
https://github.com/saltstack-formulas/napalm-install-formula

We use this formula to setup the tools.
1. Add the install state to the state top.sls:

/srv/salt/top.sls:

```
base:

  'proxy-host-*':

    - napalm_install
    - proxy-service
```

2. We are setting up IOS and Junos drivers. The salt-proxy pillar data is used to configure the proxy minions:

/srv/pillar/napalm-proxy.sls:

```
napalm:
  install:
    - napalm-junos
    - napalm-ios
salt-proxy:
  host: 10.20.30.41
  port: 4506
  fingerprint: 11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff:00:11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff:00
```

3. Link the configuration at the pillar top.sls:

/srv/pillar/top.sls:

```
{% set proxy_instances = ['swi-ex2200-c-00', 'swi-ex2300-c', 'swi-ex4300-00'] %}
base:
  'proxy-host-*':
    - napalm-proxy

{% for inst in proxy_instances %}
```

```
  '{{inst}}':
    - proxied.{{inst}}
    - switches.{{inst}}
{% endfor %}
```

4. Create a state that distributes proxy config and creates proxy instances:

/srv/salt/proxy-service.sls

```
{% set proxy_instances = ['swi-ex2200-c-00', 'swi-ex2300-c', 'swi-ex4300-00'] %}

set_proxy_master_config:
  file.managed:
    - name: /etc/salt/proxy.d/master.conf
    - source: salt://files/proxy.master.conf
    - template: jinja
    - context:
        host: {{ pillar['salt-proxy']['host'] }}
        port: {{ pillar['salt-proxy']['port'] }}
        fingerprint: {{ pillar['salt-proxy']['fingerprint'] }}

#Adds YANG support to the proxies. Needed for to perform config changes
add_napalm_yang_module:
  pip.installed:
    - pkgs:
      - napalm-yang

#Trigger setup of proxy-minion services
{% for inst in proxy_instances %}
add_salt_proxy_service_{{inst}}@{{grains['id']}}:
  service.running:
    - name: salt-proxy@{{inst}}
    - enable: True
    - watch:
      - file: set_proxy_master_config
{% endfor %}
```

5. Create a proxy configuration template. This file will be read by proxy instances. It specifies how to connect to the salt master:

/srv/salt/files/proxy.master.conf

```
master: "{{host}}"
master_port: {{port}}
master_finger: {{fingerprint}}
multiprocessing: false
pki_dir: /etc/salt/pki/proxy
```

6. Then create a file with connection/authentication data per network device:

e.g. /srv/pillar/proxied/swi-ex2200-c-00.sls

```
proxy:
  proxytype: napalm
  driver: junos
  host: 172.16.31.10
  username: napalm
  passwd: SECRET
```

…same for the other devices

```
chmod 600 /srv/pillar/proxied/swi-*.sls
```

7. Then run the state using

```
salt proxy-host-\* state.apply
```

This initially took 195s (salt time). Subsequent runs take about 23s/37s (salt time/real time).

During startup of the proxy-minion, it gathers a lot of information from the network device. On a Juniper EX2200 switch, proxy-minion startup takes about 90s. Only after that, the proxy-minion is ready to serve commands from the salt-master. First check if the proxies tried to logon on the salt-master:

```
salt-key -F
```

It’s recommended to check the new fingerprints if there is any doubt that you’re looking at the correct device (e.g. by checking /etc/salt/pki/proxy on the proxy-host, or by pre-seeding the keys). If the fingerprints seem plausible, accept the new keys:

```
salt-key -A
```

Wait a few seconds until the proxy-minions connect, then test the connectivity:

```
salt swi-\* test.ping
```

Get information

Now we have a tool to throw commands at the devices.

Let’s get the serial numbers for inventory:

salt -G ‘os:junos’ grains.get serial

```
swi-ex4300-00:
    PF112233445566
swi-ex2300-c:
    HW778899001122
swi-ex2200-c-00:
    GR334455667788
```

As mentioned in story0, we want to collect LLDP information from the devices:
salt swi-ex22 napalm.call get_lldp_neighbors –out json*

```
{

    "swi-ex2200-c-00": {

        "comment": "",

        "result": true,

        "out": {

            "ge-0/1/1.0": [

                {

                    "hostname": "zeuz-vm0",

                    "port": "ens192"

                }

            ],
            "ge-0/1/0.0": [
                {
                    "hostname": "swi-ex4300-00",
                    "port": "EX2200C Claudius desk"
                }
            ]
        }
    }
}
```

User management

Other than regular salt user states that only assure that a certain user is present or absent, the napalm user state replace the whole user config (i.e. drop any unknown accounts).

Read current users

salt swi-* users.config –out=yaml

```
swi-ex2200-c-00:
  comment: ''
  out:
    lab:
      level: 15
      password: ''
      sshkeys:

      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAA...

    napalm:

      level: 15

      password: ...

      sshkeys:

      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAA...

...
```

Configure users

Setup a state that performs the management with some Jinja+Yaml magic:

/srv/salt/switch-users.sls

```
{% if 'switch-users' in pillar %}
add_netusers_lab:
  netusers.managed:
    - users:
{% for user in pillar['switch-users'] %}
       {{ user.name }}:
         level: {{ user.level if 'level' in user else 1 }}
         password: '{{ user.password if 'password' in user else '' }}'
         sshkeys: {{ user.sshkeys if 'sshkeys' in user else [] }}
{% endfor %}
{% endif %}
```

Create a list of accounts in /srv/pillar/switch-users.sls

```
switch-users:

  - name: root
```

```
    level: 20
    password: '$1$...'
  - name: reader
    level: 1
    password: 'my-secret-plain-text-password'
    sshkeys: []
      - 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAA...'
...
```

and mention the .sls files in the corresponding top.sls files.

IMPORTANT:

- Always define a root user and its password. This is enforced by Junos; if no root user with a password is defined, the state-run will fail. We learned this the hard way. If there is no root user in the pillar, napalm will try to remove it (Junos denies). So we initially adapted the Junos templates to not delete the root user. This confused Junos even more as napalm then started sending empty commands at the devices. So we reverted everything, stepped over some transient issues where napalm didn't detect the root user and tried to add a new root user (Junos denies) – stop the proxy-minion, clear any caches, start again – and FINALLY: It just works.
- Hashed passwords are not supported. This is a limitation of the target devices (e.g. Junos supports better^H^H^H^H^H^Hdifferent encryption algorithms than IOS)
  - In our limited lab example, this limitation was mitigated by changing *plain-text-password* to *encrypted-password* in /usr/local/lib/python2.7/dist-packages/napalm/junos/templates/set_users.j2 on the proxy host. While this allows us to share nicely hashed passwords between Linux hosts and network devices, it kills vendor neutrality – must be regained using salt grain magic (e.g. grain 'os:junos')
- Here's the patch to the template to avoid deleting the 'root' user: (not needed in most scenarios)

```
root@proxy-agent-00:/usr/local/lib/python2.7/dist-packages# diff -u
```

```
napalm/junos/templates/delete_users.j2*
--- napalm/junos/templates/delete_users.j2      2019-02-15
13:49:08.728060210 +0100
+++ napalm/junos/templates/delete_users.j2.orig 2019-02-15
13:48:56.419847302 +0100
@@ -1,9 +1,7 @@
 system {
   login {
     {%- for user_name, user_details in users.items() -%}
-      {%- if user_name == 'root' -%}
-      {%- else -%}
-        {%- if user_details and (user_details.get('password') and
user_details.get('sshkeys')) -%}
+      {%- if user_details and (user_details.get('password') and
user_details.get('sshkeys')) -%}
       user {{user_name}} {
           authentication {
             {%- if user_details.get('password') -%}
@@ -20,10 +18,9 @@
             {%- endif -%}
           }
       }
-      {%- endif -%}
     {%- endfor -%}
   }
 }
```

Connect to devices using certificate-based ssh

If exchanging plain text passwords with salt or storing them in some configuration management solution is not allowed, napalm also provides means to connect to networked devices via ssh by using certificates.

On the master, add a state to /srv/salt/proxy-service.sls:

```
set_proxy_ssh_key:

  file.managed:
    - name: /etc/salt/pki/proxy/napalm

    - mode: 600

    - content_pillar: napalm:key
```

Link this state in the state top.sls as well. Add the private key to the proxy-host’s pillar:

```
napalm:

  key: |

    -----BEGIN RSA PRIVATE KEY-----

    Proc-Type: 4,ENCRYPTED

    DEK-Info: AES-128-CBC,...

 ...

    -----END RSA PRIVATE KEY-----
```

And change the proxy configuration to use a certificate instead of a password:

e.g. /srv/pillar/proxied/swi-ex2200-c-00.sls

```
proxy:

  proxytype: napalm

  driver: junos

  host: 172.16.31.10

  username: napalm

  passwd: ''

  optional_args:
    secret: ...
    key_file: /etc/salt/pki/proxy/napalm
```

The variable proxy:optional_args:secret contains the passphrase of the private key.

Make sure the user account is configured to use the corresponding pubkey:

/srv/pillar/switch-users.sls:

```
switch-users:
  - name: root
    level: 20
    password: '...'
  - name: napalm
    level: 15
    password: ''
    sshkeys:
      - 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQA... root@proxy-agent-00'
```

The inclined reader might suggest that there are still secrets being distributed using salt. But first – it is limited to salt-internal connections (no more to napalm nor to the networked device) and distributed half and half to the salt-minion (private key) and proxy-minion (the private key's passphrase) – and second, this could be further reduced by creating the certificate and credentials on the proxy-host directly (but this again would increase complexity for key distribution and redundant setups).

We had some issues with accounts having a password and sshkeys that their level was automatically set as 'unauthorized' (=level 0). Temporarily defining an empty password (i.e. disable password-based login) avoids this issue.

Changing the proxy's configuration requires a restart if it. This can be manually triggered remotely (or automatically by correctly setting up dependencies in salt).

salt proxy-agent-00 service.restart salt-proxy@swi-ex2200-c-00

You can download the full code from here.

Limitations

During the experiments with salt and napalm, the following limitations were found:

- Full official support exists only for Arista EOS, Juniper Junos and several OSes of Cisco; Multiple – sometimes feature limited or abandoned – approaches from community exist for VyOS, Cumulus, F5, Mikrotik, Brocade MLXe, Huawei CE and some more. Unfortunately, there seems to be no Brocade/Ruckus ICX or VDX switch support (that our customer of story0 is using…).
- Startup of a proxy minion creates significant CPU load on smaller devices (e.g. 20% for 90s on Juniper EX2200 running Junos 15.1R5.5).
- One could meet scaling issues on the proxy-host: In our setup, one instance of salt-proxy starts two processes that consume about 60MB and 110MB RAM. This number grows to about 60MB and 140MB after a day of idling. On the (comparably low-end) test system, an instance consumed 12m CPU time in 17h wall-clock time which makes about 1.2% CPU load per instance. In order to manage 250 networked devices, one requires 50GB RAM and 3 CPU cores for the salt-proxy instances; server-side, no significant additional resources required. Maybe that's why it's called proxy-minion and not proxy-micron.

Architectural Thoughts

The proxy-based design sounds very attractive from operations and security perspective: Only the proxy-host needs to have ingress connectivity to the network devices and egress connectivity to the salt master – e.g. a single point of contact. This greatly simplifies ACLs on firewalls, but also on network devices.

The proxy-minion concept completely hides the proxy-host itself. The network devices appear in salt as regular minions.

Redundancy: If the proxy-host dies, one loses observability and controllability of the network devices. A potential solution could be to have a cold backup proxy-host and let salt activate it using some kind of dead-man circuit (e.g. by observing the proxies' beacon). Redundancy on the salt-master side is part of salt as salt-minions and proxy-minions already provide multi-master support.

Next steps

Some topics to cover in the future:

- ntp status/config
- Interface/vlan configuration with YANG.
- Policy checking
- Patch management

Recently, an alternative tool to NAPALM has been integrated into salt: netmiko Vendor coverage seems far broader. It might be worth looking at this tool when looking for an abstraction layer for network infrastructure automation.

Conclusion

I like the (pretty basic) results, that we can now control our switches from a central tool and distribute credentials and how smooth it integrates with our existing infrastructure. Until now I only did very simple config changes, as my experience with YANG is pretty basic yet.

On the other side, I'm somewhat surprised that two major manufacturers of switches (Brocade and Huawei) are not officially supported; Brocade seems to (have?) preferred StackStorm; I didn't find any intentions of Huawei yet, other than their proprietary tools. Without YANG support, NAPALM with salt, Junos and IOS support consists of more than 40kLOC (Excluding external libs) – pretty much code for the tasks. The high memory requirements could also limit scaling in some setups, but this is a generic issue of salt. The tool's name suggests bad things will happen – luckily it didn't do so yet.

Literature

Presentations of salt-napalm by Mircea Ulinic

https://www.nanog.org/sites/default/files/NANOG68%20Network%20Automation%20with%20Salt%20and%20NAPALM%20Mircea%20Ulinic%20Cloudflare%20(1).pdf

https://2017.apricot.net/assets/files/APIC674/apricot%202017%20network%20automation%20with%20salt%20and%20napalm%20mircea%20ulinic%20cloudflare_1488177661.pdf

Napalm salt integration

https://github.com/napalm-automation/napalm-salt

Many labs on automation using virtual network hardware

https://labs.networkreliability.engineering

Salt napalm install formula:

https://github.com/saltstack-formulas/napalm-install-formula.git

Extreme/Brocade MLXe driver (get_bgp_neighbors and get_environment only)

https://github.com/ckishimo/napalm-extreme-netiron

Another approach to access Brocade devices

https://github.com/shivharis/napalm-brocade

Possible alternate to napalm:

https://github.com/internap/netman