

# Erläuterung zum Aufbau und Konzept des Playbooks

<b>Übersicht</b>	<b>2</b>
Absicherung der CD/CI	4
Zum Aufbau des Playbooks	6
Die Ansible Roles	10
Die Vorbereitungs Roles	11
Vorbereitung der Root CA-Konfiguration	12
Erstellung des Root-CA-Schlüssels	13
Erstellung des selbst signierten Root-Zertifikates	17
Die Intermediate CAs	17
Certificate Signing Request (CSR) der Intermediate CA	18
Das Signieren der Intermediate CA Zertifikats	20
Die Ansible-Roles der End-Entities	25
Die Service End-Entity	27
Die Client End-Entity	29
End-Entity Zertifikate im p12-Format für die Verwendung im Browser	30
<b>Inventory</b>	<b>32</b>
<b>Certificate Revocation List (CRL)</b>	<b>35</b>
<b>Debugging-Modus</b>	<b>36</b>

# Übersicht

Um klar zu machen welches Problem dieses Playbook lösen will, hier ein Diagramm, das verdeutlichen soll, wie bei einer mehrstufigen PKI-CA-Hierarchie die einzelnen Komponenten in der richtigen Reihenfolge die richtigen Schritte tun müssen.

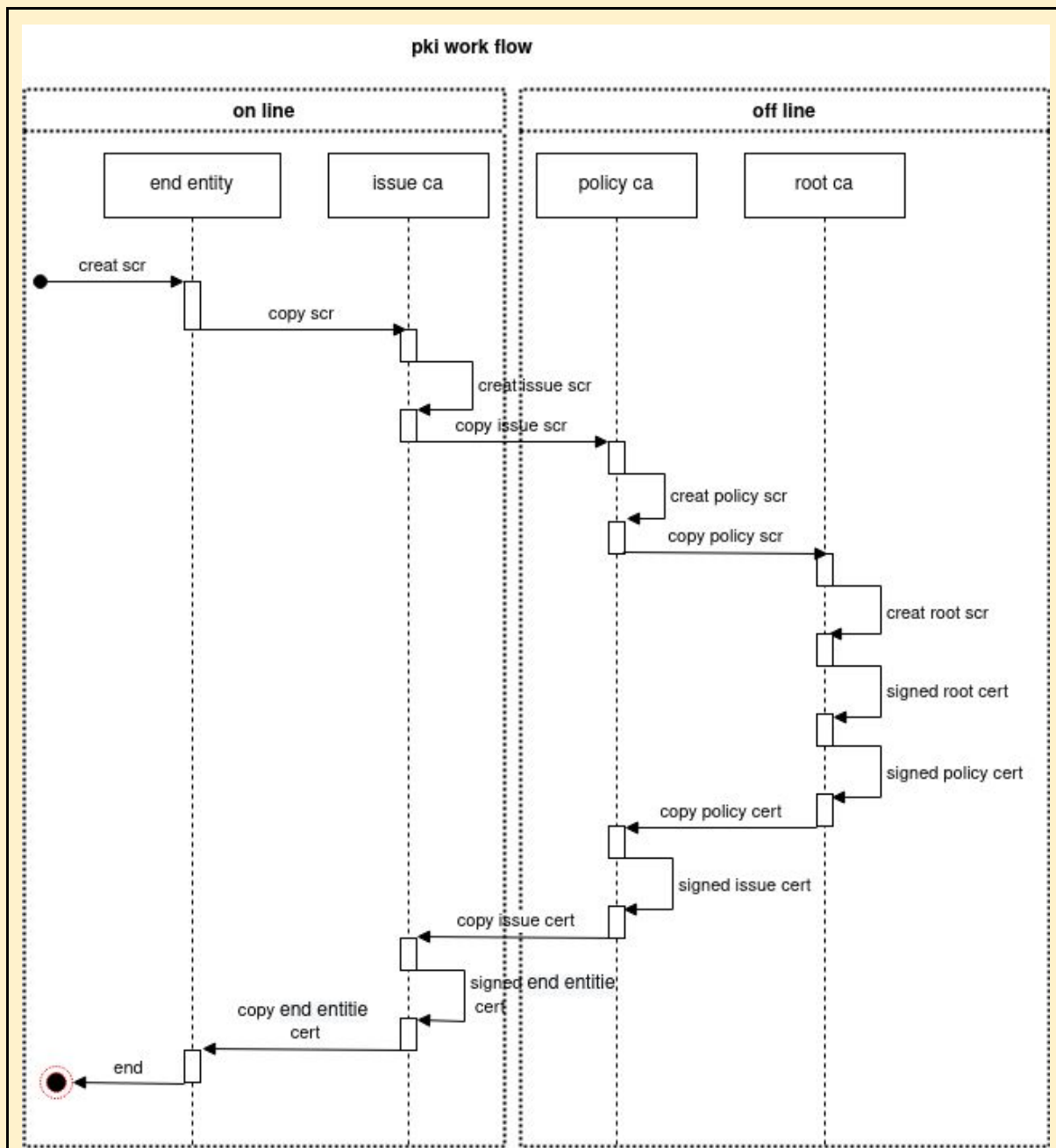
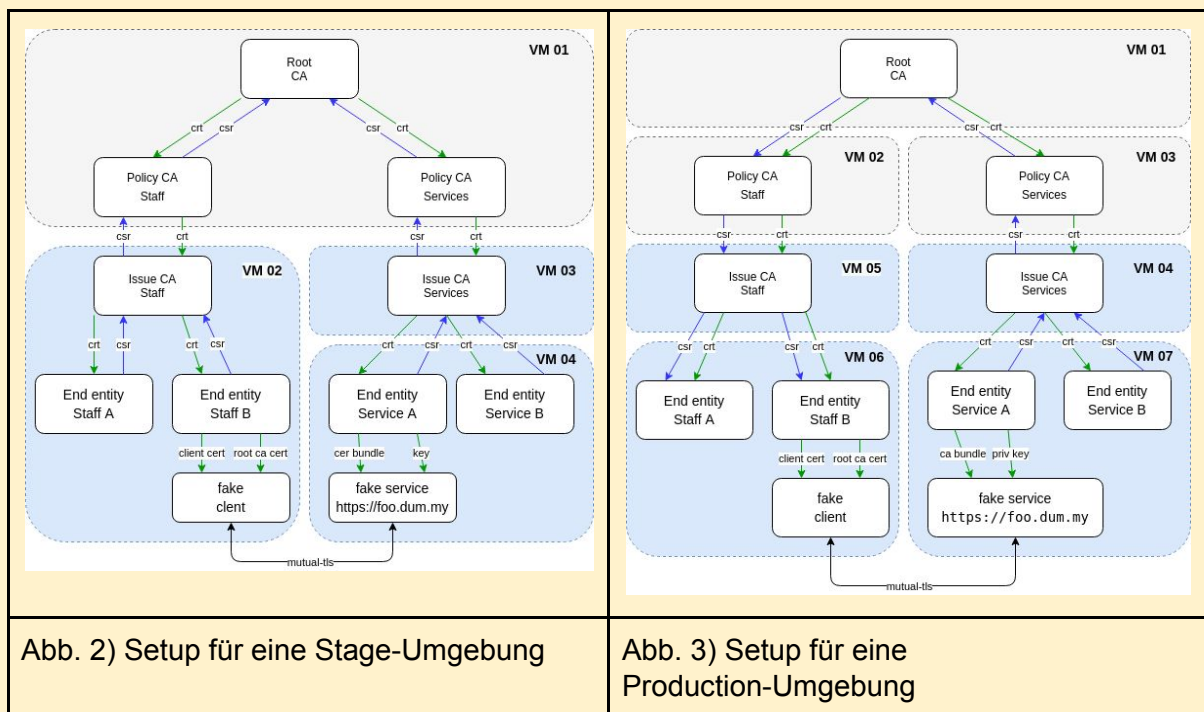


Abb 1) Ablaufdiagramm eines Aufbau einer "chain of trust".

Damit eine End entity sich ein Zertifikat ausstellen lassen kann, müssen alle übergeordneten CAs das für sich selber auch alle getan haben. So entsteht dann die s.g. "Chain of Trust". In diesem Fall sind das 14 Schritte 3 Hierarchieebenen. Um die Sicherheit zu erhöhen, ist ein Teil der PKI-CAs offline (rechts im Bild) und muss zusätzlich bei Bedarf online (eingeschaltet) werden.

Eine Automation dieser zahlreichen Schritte lässt sich mit Ansible erreichen, wie in diesem PoC gezeigt werden wird.

Sicherheit ist immer mit Aufwand an Arbeit und Ressourcen, und letzten Endes mit Kosten verbunden. Das Resultat einer Kosten-Nutzen-Bewertung fällt, je nach Anwendungsfall, unterschiedlich aus. Deswegen wurde ein PoC erstellt, der offen lässt, auf wie viel VMs die PKI-CAs aufgeteilt wird.



Im Diagrammen ist der selbe PKI-Baum zwei mal zusehen. Ein mal auf 4 VMs und ein mal auf 7 VMs verteilt. Das linke Szenario stellt die Testumgebung (Stage) dar. Die rechte Graphik zeigt die Live-Umgebung (production). Dies lässt sich mit Ansible leicht über unterschiedliche s.g. Inventories abbilden. Abb. 4 zeigt das inventory von Stage und Abb. 5 das von Production.

<pre> inventories &gt; staging &gt; group_vars &gt; ! pki.yml 1  --- 2 3  debug_output:.....true 4  ignore_errors:.....yes 5 6  vm_01:....."51.116.112.42" 7  vm_02:....."51.116.175.33" 8  vm_03:....."51.116.175.126" 9  vm_04:....."51.116.175.166" 10 11 12  root_ca_ip:....."{{ vm_01 }}" 13  policy_ca_staff_ip:....."{{ vm_01 }}" 14  policy_ca_service_ip:....."{{ vm_01 }}" 15  issue_ca_staff_ip:....."{{ vm_02 }}" 16  issue_ca_service_ip:....."{{ vm_03 }}" 17  foo_dum_my_ip:....."{{ vm_04 }}" 18  bar_dum_my_ip:....."{{ vm_04 }}" 19  baz_dum_my_ip:....."{{ vm_04 }}" 20  jane_doe_ip:....."{{ vm_02 }}" 21  john_doe_ip:....."{{ vm_02 }}" </pre>	<pre> inventories &gt; production &gt; group_vars &gt; ! pki.yml 2 3  debug_output:.....true 4  ignore_errors:.....yes 5 6  vm_01:....."20.52.41.243" 7  vm_02:....."20.52.41.152" 8  vm_03:....."20.52.41.88" 9  vm_04:....."20.52.41.223" 10 vm_05:....."51.116.186.241" 11 vm_06:....."51.116.187.148" 12 vm_07:....."20.52.41.92" 13 14 15  root_ca_ip:....."{{ vm_01 }}" 16  policy_ca_staff_ip:....."{{ vm_02 }}" 17  policy_ca_service_ip:....."{{ vm_03 }}" 18  issue_ca_staff_ip:....."{{ vm_04 }}" 19  issue_ca_service_ip:....."{{ vm_05 }}" 20  jane_doe_ip:....."{{ vm_06 }}" 21  john_doe_ip:....."{{ vm_06 }}" 22  foo_dum_my_ip:....."{{ vm_07 }}" 23  bar_dum_my_ip:....."{{ vm_07 }}" 24  baz_dum_my_ip:....."{{ vm_07 }}" </pre>
Abb. 4)	Abb. 5)

## Absicherung der CD/CI

Bevor wir in die Details des Ansible-Playbooks einsteigen, hier noch eine Erörterung, wie man die Automatisierung selbst absichert, damit die ganze Arbeit mit der PKI umsonst war, weil die CI/CD-Pipeline gehackt wurde.

## The big picture of certificate signing process

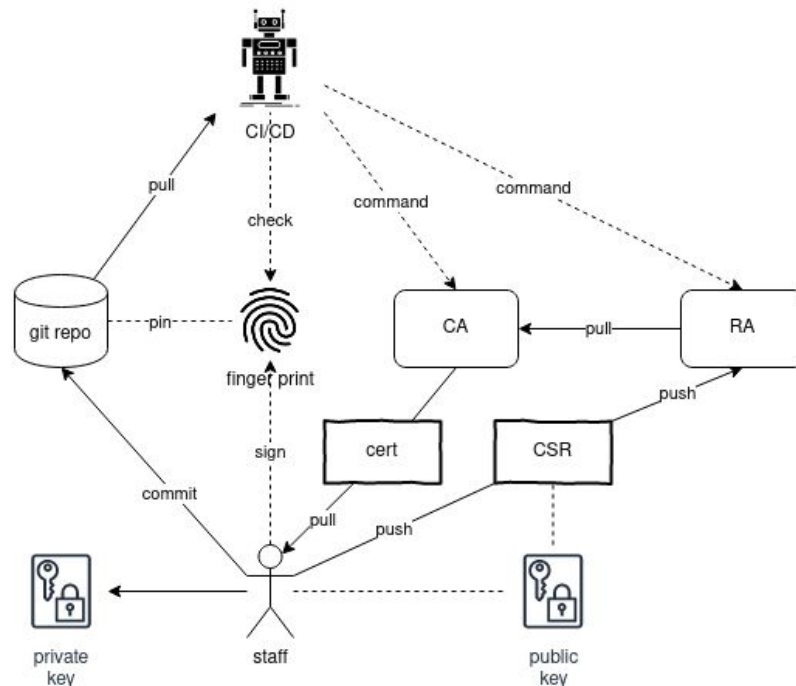


Abb. 6) Absicherung der CI/CD

Dreh- und Angelpunkt ist hier (wenig überraschend für “Infrastructure-as-Code” bzw. “GitOps”) das Git-Repo. Als erste Sicherheitsmaßnahme wird ein Tool wie GitLab oder Gitea verwendet, um den Zugang zu beschränken. Der nächste Schritt ist, alle Commits von dem Bearbeiten signieren zu lassen. Dann kann ein CI/CD-Tool wie Jenkins, Concourse-CI, AWX oder GitLab-CI-Runner beim Auschecken des Ansible-Playbooks den Fingerprint des Commits prüfen. Im Ansible-Playbook sieht der schritt wie in Abb. 7 aus.

```

roles > check_git_sign > tasks > ! main.yml
8  - name: ..... Check the fingerprint of the commit sign
9    git:
10     repo: ..... "{{ pki_repo }}"
11     dest: ..... "{{ pki_repo_dest }}"
12     version: ..... "{{ pki_git_tag }}"
13     force: ..... yes
14     gpg_whitelist: ..... "{{ pki_gpg_whitelist }}"
15     verify_commit: true
  
```

Abb. 7)

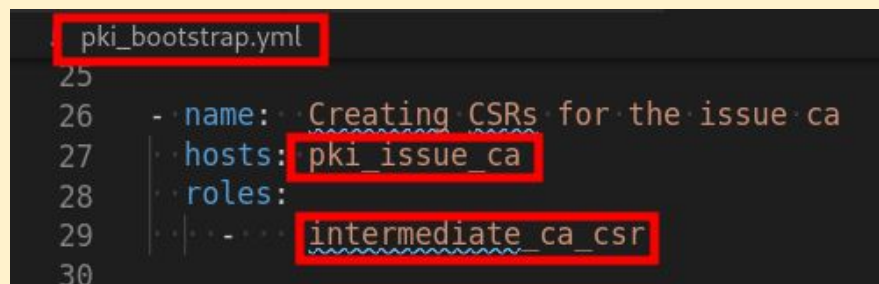
Das Certificate Signing Request (CSR) könnte man auch in Git einchecken. In diesem Fall ist es aber so, dass die Zertifikatnehmer und -geber CSR und Zertifikate über http austauschen. Beide haben eine Nginx, über die sie beides zum Download anbieten.

Public Key, CSR und signiertes Zertifikat stellen kein Geheimnis dar, das geschützt werden muss (im Gegensatz zum private Key, der seinen Eigentümer niemals verlassen sollte). Wichtig ist auch, dass der Zertifikatsaussteller (CAs) überprüfen kann, dass die Identität im CSR mit den darin enthaltenen privaten Key identisch ist, also dass ein Antragsteller sich nicht ein Zertifikat für jemanden anderen ausstellen lassen kann. In unserem PoC vertrauen die CAs der IP. Wenn man nicht den IPs trauen kann und auf Domains geht, bräuchte man zur Absicherung https. Dann hätte man aber ein Henne-Ei-Problem und muss sich einen anderen Weg überlegen, wie man die Glaubwürdigkeit des CSR absichert, z.B. indem man das CSR doch in git eincheckt. Unberührt davon, können die CAs aber weiterhin, ihre Zertifikate den Eigentümern über http zum Download anbieten, wenn die CA prinzipiell für diese erreichbar ist, -wovon dieses Beispiel ausgeht.

## Zum Aufbau des Playbooks

In dem PoC finden sich fünf Playbooks: *azure\_bootstrap.yml* und *azure\_destroy.yml* sind zwei Hilfs-Playbooks, mit dem man VMs in der Azure-Cloud erstellen und wieder löschen kann. Diese sind nur als Hilfe gedacht, um möglichst schnell mit dem eigentlichen Playbook starten zu können. Das Playbook *git\_check.yml* ist nur als Beispiel gedacht wie man die Integrität der Git-Commits prüfen kann. Das eigentliche Playbook ist das *pki\_bootstrap.yml*. Dies konfiguriert die VMs so, dass ein Baum von CAs entsteht, wie sie in Abb. 2 und 3 zu sehen ist. Das Playbook *pki\_reset.yml* ist ein weiteres Hilfs-Tool um die generierten Daten in den CAs zu löschen, wenn man sie neu generieren lassen will.

Ansible arbeitet prinzipiell **sequenziell**. In dem Playbook sieht man welche Host-Groups mit welchen Roles verknüpft werden (Abb 8). Dadurch lassen sich die Schritte aus Abb 1, in der Datei *pki\_bootstrap.yml* in derselben Reihenfolge wiederfinden.



```
pki_bootstrap.yml
25
26 - name: Creating CSRs for the issue ca
27   hosts: pki_issue_ca
28   roles:
29     - intermediate_ca_csr
30
```

Abb 8) Hier wird im Playbook *pki\_bootstrap.yml* die Host-Group *pki\_issue\_ca* mit der Rolle *intermediate\_ca\_csr* verknüpft

```

inventories > staging > ! hosts.yml
9      ······root-ca-01.dum.my:
10     ······pki_policy_ca:
11     ······hosts:
12     ······service-policy-ca-01.dum.my:
13     ······staff-policy-ca-01.dum.my:
14     ······pki_issue_ca:
15     ······hosts:
16     ······service-issue-ca-01.dum.my:
17     ······staff-issue-ca-01.dum.my:
18     ······pki_end_entity:
19     ······children:
20     ······end_entity_service:
21     ······hosts:
22     ······foo.dum.my:
23     ······bar.dum.my:

```

Abb 9) In dem Inventory-File *hosts.yml* kann man sehen, welche VMs der Gruppe *pki\_issue\_ca* zugeordnet ist.

Damit Ansible die einzelnen VMs per ssh ansteuern kann, müssen IP- oder Domain-Adressen für die einzelnen VMs hinterlegt werden. Dies geschieht in der Datei *inventories/staging/group\_vars/pki.yml* bzw *inventories/production/group\_vars/pki.yml* je nachdem, ob man das Verhalten Production- oder Staging-Umgebung verändern will. Jede Umgebung hat ihre eigenes Inventory (Abb 10).



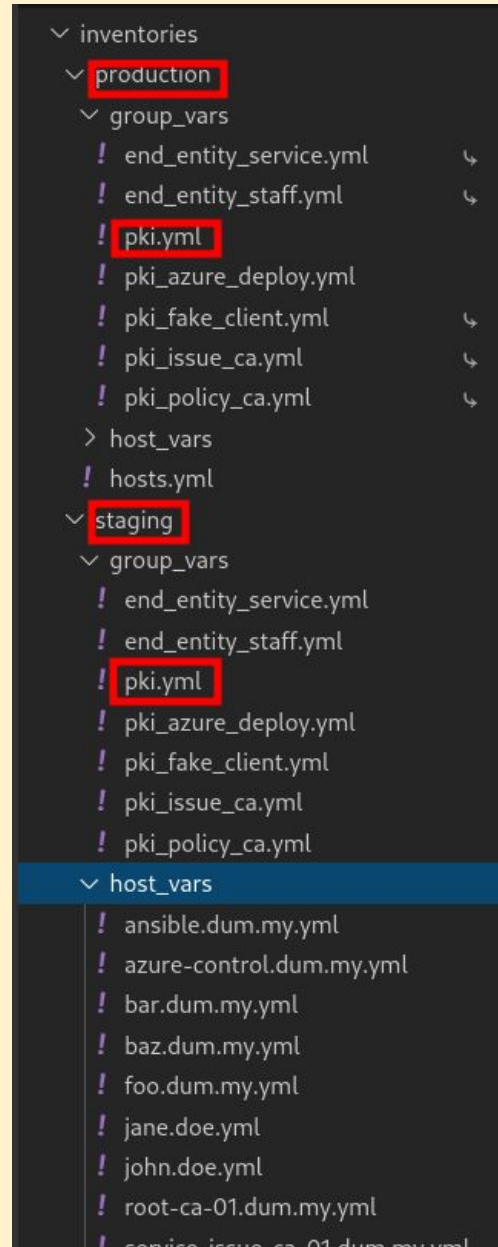
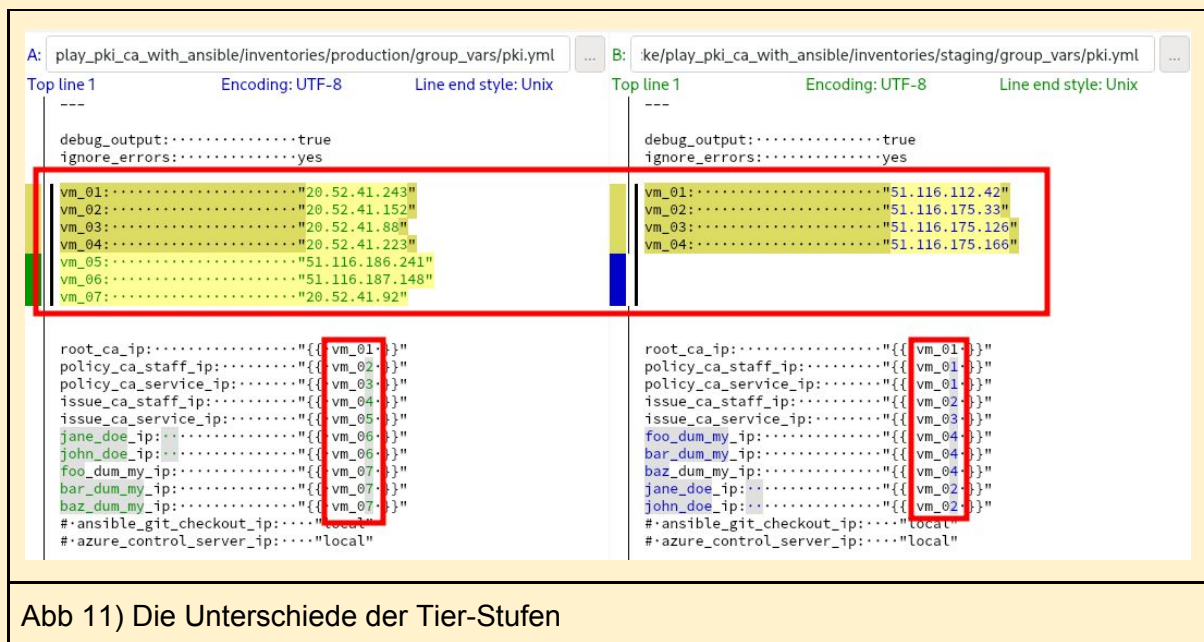


Abb 10) Stage- und Production-Umgebung haben ihre eigenen Inventory-Verzeichnisse.

An dieser Stelle noch der Hinweis: die meisten Dateien im Inventory Production sind Links auf Dateien im Inventory-Verzeichnis von Stage. Dadurch wird Redundanz vermieden, denn die Group- und Host-Konfiguration ist fast gleich. Die einzigen Unterschiede zwischen den Tier-Stufen sind in der Datei *inventories/production/group\_vars/pki.yml* bzw. *inventories/staging/group\_vars/pki.yml* aggregiert.





Die Unterschiede in der Datei *pki.yml* sind gering. Lediglich die Anzahl der VMs und die Verteilung der IPs auf die Hosts (Siehe Abb 11 unterscheiden sich). In diesem Playbook teilen sich einige Hosts die VMs und haben deshalb die selbe IP-Adresse. Eigentlich repräsentiert in einem Ansible-Playbook ein Host tatsächlich eine einzelne VM oder Maschine. Hier repräsentiert ein Host jedoch ein Element einer PKI. Zudem können hier mehrere PKI-Element auf einer VM beheimatet sein und teilen sich deshalb eine VM mit ihrer IP. So ist z.B. die Root-CA auf der gleichen VM/IP wie die Policy-CA.

# Die Ansible Roles

Nun zu den Ansible-Roles. Zunächst die tabellarische Übersicht:

Role Name	PKI-Komponente	Kommentar
check_git_sign	%	klont und prüft ein Git-repo
deploy_azure_vms	%	erstellt VMs in der Azure Cloud
destroy_azure_vms	%	entfernt die erstellten VMs wieder in der Azure Cloud
end_entity_csr	end entity	erstellt Ein CSR für eine End Entity
fake_client	end entity	simuliert eine End Entity in Form einer Person die ein Service aufruft
fake_service	end entity	simuliert eine End Entity in Form eines Web-Service
intermediate_ca_csr	intermediate ca	erstellt eine CSR für eine generische Intermediate CA
intermediate_ca_signing	intermediate ca	simuliert die Signaturtätigkeit einer generischen Intermediate CA
pki_reset	CAs	löscht alle erstellten generischen Daten, Keys, Signaturen, CSRs, usw.
pre_config	CAs	bereitet die notwendigen Dateien und Verzeichnisse der CAs vor
pre_update_and_install	%	bereitet die VMs auf ihre Aufgabe vor
root_ca_self_signing	root ca	erstellt eine Root CA mit selbst signiertem Zertifikat
set_special_host_facts	CAs	um Redundanz zu vermeiden, werden hier einige Variablen generisch erstellt
Tabelle 1)		

Wir gehen die Rules nicht alphabetisch durch, sondern nach ihren Funktionen und zwar “top-down” wie in der Abb 2 bzw Abb 3 gezeigt von der root-CA absteigend! Das ist nicht die chronologische Reihenfolge die Abb 1 gezeigt wird. Auf diese Weise lässt sich besser

verdeutlichen, in welchen Bezug die einzelnen PKI-Komponenten stehen, und wie dies im Playbook abgebildet wird.

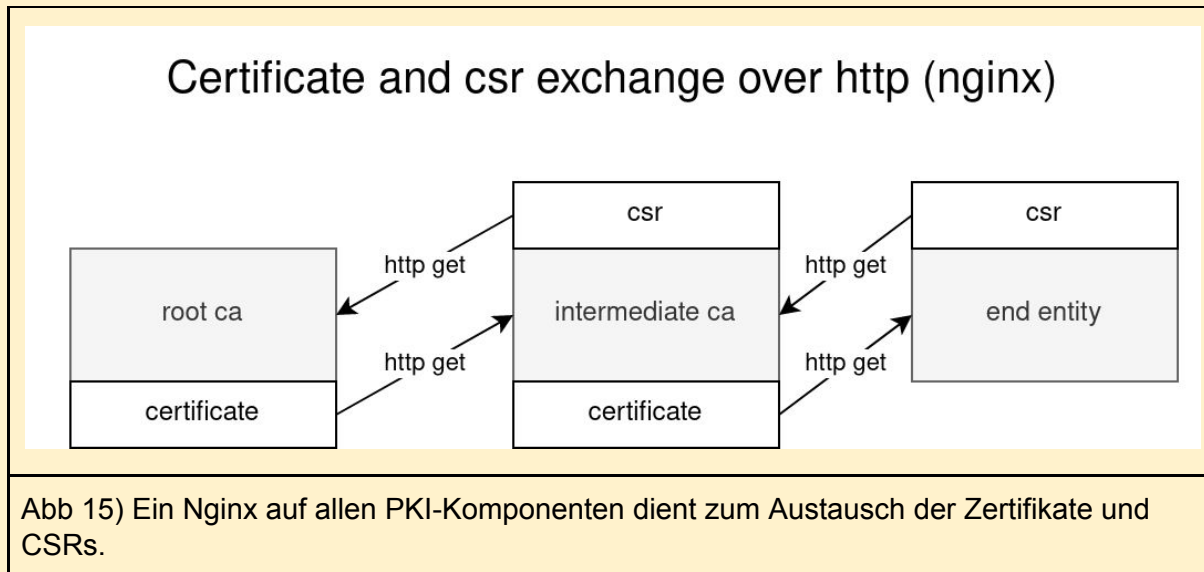
## Die Vorbereitungs Roles

Bevor die eigentlich PKI aufgebaut wird, werden noch die Roles *set\_special\_host\_facts*, *pre\_update\_and\_install* und *pre\_config* ausgeführt. Die *set\_special\_host\_facts* wird weiter unten erläutert. *pre\_update\_and\_install* werden die Systeme auf den aktuellen Softwarestand gebracht und notwendige Paket installiert wie z.B. der Nginx. Die Role *pre\_update\_and\_install* ist im Vergleich zu den anderen sehr langsam. Das hat unter anderem den Grund, dass diese Schritt nicht parallel auf allen Hosts gleichzeitig ausgeführt wird. Dass die PKI-Komponenten flexibel auf die vorhandenen VMs aufgeteilt werden können, kann es sein, da mehrere Komponenten auf einer VM erstellt werden. Das weist diese Role aber nicht. Wenn man sie nicht daran hindern würde, kann es sein, dass sie versucht auf der gleichen VM den Update- und Install-Prozess mehrmals gleichzeitig zu starten. Das hat zumindest bei mir dazu geführt, dass die Paketdatenbank unter Ubuntu blockiert wurde. Um das zu verhindern, wird diese Role nur nacheinander auf den Hosts ausgeführt.

```
! pki_bootstrap.yml
10 - name: Update and install needed software
11   hosts: pki
12   serial: 1
13   roles:
14     - pre_update_and_install
```

Abb 14) Zeile 12 sorgt dafür, dass immer nur ein Host gleichzeitig konfiguriert wird.

Die Role *pre\_config* konfiguriert auf allen PKI-Host einen Nginx über den die PKI-Komponenten dann später ihre CSRs und Zertifikate austauschen.



## Vorbereitung der Root CA-Konfiguration

Es gibt einige Dinge die bei einer root CA identisch mit anderen CAs sind. In der Datei *root\_ca\_self\_signing/tasks/init\_dir.yml* wird eine Verzeichnisstruktur aufgebaut, die für die root CA notwendig ist. Die darin generierten Verzeichnisse und Dateien sind dieselben wie bei der Intermediate CA. Das sind:

- ein Verzeichnis in dem der Private Schlüssel und andere Dinge abgelegt wird
- eine Index-Attributes-Datei
- eine Datei für die CRL Nummer. CRL steht für "*Certification Revocation List*". Also die Zertifikatsperrliste, die wir aber in dem PoC nicht verwenden
- eine openssl-Konfiguration die aus einem Template erstellt wird

Alles Verzeichnis- und Dateinamen sind generisch und können in der Datei *roles/set\_special\_host\_facts/tasks/main.yml* angepasst werden. Damit das auf diese einfache Weise möglich ist, gibte es die role *set\_special\_host\_facts*.

```

roles > set_special_host_facts > tasks > ! main.yml
3  - name: ..... Set individual pathes and other things for root_ca
4    block:
5      - set_fact:
6        pki_private_dir: ..... "{{ pki_workspace }}/{{ pki_root_ca.common_name }}"
7
8      - set_fact:
9        pki_opensslconfig: ..... "{{ pki_private_dir }}/openssl.cnf"
10       pki_priv_key: ..... "{{ pki_private_dir }}/private/{{ pki_root_ca.common_name }}.key.pem"
11       pki_serial_file: ..... "{{ pki_private_dir }}/serial"
12       pki_index_file: ..... "{{ pki_private_dir }}/index"
13       pki_crlnumber_file: ..... "{{ pki_private_dir }}/crlnumber"
14       pki_cert_file: ..... "{{ pki_ca_dir }}/{{ pki_root_ca.common_name }}.crt.pem"
15       pki_index_attr_file: ..... "{{ pki_private_dir }}/index.attr"
16       pki_bundle_file: ..... "{{ pki_publication_dir }}/{{ pki_root_ca.common_name }}.bundle.pem"
17     when: ..... root_ca is defined

```

Abb 12) Der CA bzw. End entity-Name ist im Anfang des Datei-Pfad codiert. Deshalb war es die eleganteste Lösung, die Pfade in einer extra Role generieren zu lassen. Hierdurch bleiben die Host-Konfigurationen unterhalb von *inventories/production/host\_vars* schlank und übersichtlich. Über die Konditionen, wie hier in Zeile 18, werden die Besonderheiten zwischen Root-CA, Intermediate-CA Und End-Entity abgebildet.

Alle CAs müssen sicherstellen, dass sie keine Seriennummern doppelt verwenden. Dies geschieht durch die Verwendung des Flag “-rand\_serial” (Abb 13 Zeile 10). Hierbei handelt es sich nicht, wie der Name “serial” vielleicht vermuten lässt, um eine fortlaufende Nummer. Es ist vielmehr eine zufällige (random) Kennung. Es kann somit keine Rückschlüsse darauf gezogen werden wie viele Zertifikate damit erstellt wurden. Das erhöht nicht unbedingt die Sicherheit, aber in manchen Fällen den Datenschutz.

```

roles > intermediate_ca_signing > tasks > ! signed_certificate.yml
3  - name: ..... Generate and sign immediate CA certificate
4    shell: |
5      openssl ca \
6        -config {{ pki_opensslconfig }} \
7        -days {{ item.value.not_after_days }} \
8        -notext \
9        -batch \
10       -rand_serial \
11       -in {{ pki_csr_local_dir }}/{{ item.key }}.csr.pem \
12       -out {{ pki_publication_dir }}/{{ item.key }}.crt.pem

```

Abb 13)

## Erstellung des Root-CA-Schlüssels

In der Datei *roles/root\_ca\_self\_signing/tasks/create\_private\_key.yml* wird der private Schlüssel der Root-CA erstellt. Das geschieht durch einen Aufruf des tools openssl (siehe Abb 14). Ansible verfügt auch über openssl-Module:

- openssl\_certificate
- openssl\_privatekey
- openssl\_csr

Sie ermögliche den Umgang mit OpenSSL auf eine abstrakte Weise. Dadurch vermeidet man z.B. den überbordenden Umgang mit Dateien und Verzeichnissen, die OpenSSL normalerweise braucht, um arbeiten zu können. Die Kehrseite ist aber, dass PKI ein sehr komplexes Thema ist, und das es nicht immer hilfreich ist, das Dinge weg abstrahiert werden. Die reichhaltige OpenSSL-Dokumentation, die im Internet zu finden ist, hilft meist bei den Ansible-Modulen nicht weiter.

Wenn man das OpenSSL-Tool in Ansible direkt aufruft, kann man einzelne Schritte zur Fehlersuche auch noch per Hand ausführen (quasi als "Bypass"). Zudem es sind noch mehr Parameter zugänglich an denen man "schrauben" kann, aber alles zu dem Preis, dass man sehr viel mit Dateien und Verzeichnissen jonglieren muss.

```
roles > root_ca_self_signing > tasks > ! create_private_key.yml
3  - name: Generate an root ca private key with the default values
4    shell: |
5      openssl genrsa \
6        -out {{ pki_priv_key }}
7    args:
8      chdir: "{{ pki_private_dir }}"
9      creates: "{{ pki_priv_key }}"
10
```

Abb 14) Zeile 9 sorgt dafür, dass Ansible prüfen kann, ob der Schritt ausgeführt werden muss oder ob die Datei schon erzeugt wurde und der Schritt übersprungen werden kann. Das Playbook wird erst ein neuen Schlüssel erstellen, wenn der Alte zuvor gelöscht wurde.

OpenSSL ist eigentlich darauf ausgelegt, das man mit dem Tool interaktiv arbeitet. Ruft man OpenSSL einfach so auf, entwickelt sich ein munteres Frage-Antwort-Spiel in Dialog-Form. Würde es sich um eine GUI handeln, würde man von einem *wizard* or *setup assistant* sprechen.

Das Konzept stammt noch aus den 90ern, als Admins jedes System noch mit der Hand einzeln hochgezogen haben. Heute ist dass hinderlich und in Zeiten von *CI/CD* und *Infrastructure-as-Code* schon fast befremdlich. Um das sperrige Verhalten zu unterbinden, ist es notwendig ein OpenSSL-Configurationsfile zu verwenden.



```

roles > intermediate_ca_csr > tasks > ! init_dir.yml
48 - name: ..... Create openssl.cnf
49   template:
50     src: ..... openssl.cnf
51     dest: ..... "{{ pki_opensslconfig }}"
52

```

Abb 16)

Je nach Kontext (Root-CA, Intermediate-CA oder End-Entity ) sieht die Konfiguration etwas anders aus. In der Datei *init\_dir.yml* findet sich deshalb ein Schritt, der diese Konfiguration aus einem Template erstellt (Siehe Abb 16). Die Distributionen legen ihrem OpenSSL-Paketen immer Beispiel- und Default-Konfigurationen bei. Diese sind unterschiedlich aufgebaut und nur bedingt kompatibel miteinander. Das ist ein ähnliches Ärgernis wie früher mit den Init-Scripts bevor es Systemd gab. Das führte dann zu unangenehmen Sonderlocken wie in Abb 17.

```

roles > intermediate_ca_csr > templates > openssl.cnf
3
4 {% if ansible_distribution != "Ubuntu" %}
5 openssl_conf ..... = default_modules
6
7 [ default_modules ]
8 ssl_conf ..... = ssl_module
9
10 [ ssl_module ]
11 system_default ..... = crypto_policy
12
13 [ crypto_policy ]
14 .include ..... = "{{ pki_workspace }}/opensslcnf.config
15
16 {% endif %}
17
18 [ new_oids ]

```

Abb 17) Dieser Block wird nur eingefügt, wenn es sich nicht um eine Ubuntu handelt.

Die OpenSSL-Konfiguration ist über 130 Zeilen lang. Einer der wichtigsten Abschnitte ist in Abb 18+19 zu sehen. Der Wert in *"basicConstraints"* entscheidet darüber, ob ein Zertifikat für eine CA ausgestellt wird, die ihrerseits Zertifikate ausstellen kann, oder ob es sich um eine End-Entity handelt, die keine Zertifikate ausstellen darf. Darüber hinaus kann mit dem Parameter *"pathlen"* die maximale Länge der Chain-of-Trust vorbestimmt werden. Das ist wichtig, damit das Recht von CAs Aufgaben an andere CAs zu delegieren nicht dazu führen, dass die Kette unkontrollierbar lang wird.



```
roles > end_entity_csr > templates > openssl.cnf
```

```
95 [ v3_req ]  
96 basicConstraints = CA:FALSE  
97 keyUsage = {{ end_entity.key_usage_req }}  
98
```

```
roles > root_ca_self_signing > templates > openssl.cnf
```

```
93  
94 [ v3_req ]  
95 basicConstraints = critical, CA:true, pathlen:5  
96 keyUsage = nonRepudiation, digitalSignature, keyEncipherment  
97
```

Abb 18+19)

Standardmäßig würde das OpenSSL-Tool, aus Sicherheitsgründen, den privaten Schlüssel mit einem Passwort schützen. Das war in den 90er und 2000ern in den Anfängen der Virtualisierung x86er-Welt sicher eine gute Idee. Aber heute, in einer hoch automatisierten CI/CD-Welt, ist das aber kein Zugewinn an Sicherheit, da dieses Passwort auch aufwendig mit anderen Tools verwaltet werden müsste.



Abb 40) HSM von Yubi

Zur Absicherung sollten wir besser eine dezidierte VM mit verschlüsseltem Volumen verwenden oder noch besser ein *Hardware Security Module (HSM)*. Diese gibt es schon für ca. 650\$ von Yubi (Abb 40). Cloud-Anbieter haben auch HSM-as-Service im Angebot. Dies sind bei längerer Laufzeit aber bedeutend teuer in Unterhalt. Um nun nun OpenSSL die Passwort-Verwendung auszutreiben, ist die Entfernung von *challengePassword*, in der Zeile 86 in Abb 17 notwendig.

```

roles > root_ca_self_signing > templates > openssl.cnf
84
85 [ req_attributes ]
86 #challengePassword → ..... = A challenge password
87 unstructuredName → ..... = An optional company name
88

```

Abb 17) Der private Schlüssel wird **nicht** mit einem Passwort abgesichert.

## Erstellung des selbst signierten Root-Zertifikates

In Datei `roles/root_ca_self_signing/tasks/create_self_signed_cert.yml` kommen wir nun zu dem Erstellen des selbst signierten Root-Zertifikates. Hier sehen wir jetzt in Zeile 6, Abb 20, das die zuvor generierte Konfiguration verwendet wird, um zu verhindern, dass OpenSSL in den Dialog-Modus fällt. Darüber hinaus, wird in Zeile 8 die Gültigkeitsdauer angegeben (die bei der Root-CA sehr viel höher ist, als bei den anderen CAs); in Zeile 12 wird der zuvor generierte private Schlüssel übergeben; und in Zeile 13 ist der Speicherort des nun generierten Root-Zertifikates gesetzt. Auf die übrigen Parameter kann an dieser Stelle nicht weiter eingegangen werden (Sie hier zu <https://www.openssl.org/docs/manpages.html>).

```

roles > root_ca_self_signing > tasks > ! create_self_signed_cert.yml
3 - name: .....Generate a Self Signed OpenSSL root certificate
4   shell: |
5     .....openssl req \
6     .....-new \
7     .....-x509 \
8     .....-days {{ pki_root_ca.not_after_days }} \
9     .....-nodes \
10    .....-config {{ pki_opensslconfig }} \
11    .....-extensions v3_ca \
12    .....-key {{ pki_priv_key }} \
13    .....-out {{ pki_cert_file }}
14   args:
15     chdir: ..... "{{ pki_private_dir }}"
16     creates: ..... "{{ pki_cert_file }}"
17     register: ..... shell_result
18     ignore_errors: ..... "{{ debug_output }}"

```

Abb 20)

## Die Intermediate CAs

Jetzt haben wir eine Root-CA und können uns damit Zertifikate für die Intermediate CAs ausstellen. Vieles an einer Intermediate CA ist ähnlich zu einer Root-CA, aber es gibt auch

entscheidende Unterschiede. Darüber hinaus unterscheiden sich die Konfigurationen der Intermediate CAs auch untereinander durch ihre Aufgaben. In diesen PoC haben wir vier unterschiedliche Intermediate CAs.

Sie unterscheiden sich:

- in ihrem Platz in der Hierarchie der PKI
- darin ob sie (temporär) offline sind
- wie lange ihre Zertifikate gültig sind (und damit, wie lange die Zertifikate gültig sind, die sie ihrerseits ausstellen)
- ob sie Zertifikate für andere CAs ausstellen oder für End-Entity
- und ob sie mit Zertifikaten für Menschen (*Staff*) oder für Software (Services) zu tun haben

Die Role(s) in den Ansible Playbook ist so generisch gestaltet, dass sie all dies Unterschiede abbilden kann. Das Charakteristikum einer Intermediate CA ist, dass sie als Zwischenglied der *Chain of Trust* in der PKI-Hierarchie zwei Richtungen kommuniziert:

- sie lässt sich von einer höheren Instanz beglaubigen - dies wird durch die Role *intermediate\_ca\_csr* abgebildet
- sie beglaubigt ihrerseits eine oder mehrere niedrigere Instanzen - das wird in der Role *intermediate\_ca\_signing* abgebildet

## Certificate Signing Request (CSR) der Intermediate CA

Damit eine Intermediate CA gültige bzw. akzeptierte Signaturen ausstellen kann, muss sie ihrerseits zuvor ihr Zertifikat signieren lassen. Der erste Schritt dazu, ist die Erstellung einer Zertifikatsignierungsanforderung: Certificate Signing Request (CSR). Dies geschieht in der Role *intermediate\_ca\_csr*.

Das CSR besteht aus den Informationen des Zertifikat-Inhabers und seinem öffentlichen Schlüssel. Die CA Beglaubigt mit ihrer Signatur die Angaben zu dem Besitzer. Jeder, der den privaten Schlüssel besitzt, kann sich als derjenige ausweisen, der im Zertifikat genannt wird. Deshalb ist es sehr wichtig, dass der private Schlüssel niemals in die falschen Hände gerät. Das Zertifikat selbst kann nicht missbraucht werden und ist deshalb auch nicht weiter schützenswert. Dasselbe gilt für das CSR. Auch das kann nicht missbraucht werden. Deswegen wird das CSR von der Intermediate CA per Nginx und http zum Download bereitgestellt. Wichtig ist natürlich, dass die CA, die der Intermediate CA das Zertifikat signiert, darauf vertrauen kann, dass das CSR nicht durch ein Man-in-the-middle-attack, korrumpiert wurde (um zum Beispiel den falschen Schlüssel unterzuschieben). Entweder gilt das Netzwerk, mit dem DNS und die IP-Adresse als sicher, oder die Domain des Nginx braucht ein Zertifikat und muss https verwenden. Dann haben wir ein Henne-Ei-Problem: Die CA braucht ein gültiges Zertifikat um ein CSR zu erstellen, aber die CA kann keine CSR

erstellen, weil sie kein gültiges Zertifikat hat. Dieses Problem wird in diesem PoC ausgeklammert.

```
$ diff intermediate_ca_csr/templates/openssl.cnf root_ca_self_signing/templates/openssl.cnf

< policy                = policy_anything
> policy                = policy_match
---
< basicConstraints      = critical,CA:true,pathlen:{{ pki_intermediate_ca.pathlen }}
> basicConstraints      = critical,CA:true,pathlen:5
---
< keyUsage              = {{ pki_intermediate_ca.key_usage_req }}
> keyUsage              = nonRepudiation, digitalSignature, keyEncipherment
```

Abb 21)

Die Initialisierung der Verzeichnisse und Dateien sind bei der Root- und Intermediate CA fast identisch. Unterschiede gibt es beim Template für die OpenSSL-Config (openssl.cnf), siehe Abb 21. Hier wird die Verwendungsart der Zertifikate generischer kontrolliert und über die host\_vars-Dateien definiert (Siehe Abb 22).

```
inventories > staging > host_vars > ! service-issue-ca-01.dum.my.yml
8  pki_intermediate_ca:
9      common_name: "service-issue-ca-01.dum.my"
10     email_address: "service-ca@dum.my"
11     country_name: "{{ pki_country_name }}"
12     state_or_province_name: "{{ pki_state_or_province_name }}"
13     city_name: "{{ pki_city_name }}"
14     organization_name: "{{ pki_organization_name }}"
15     organizational_unit_name: "Service department"
16     not_after_days: "180"
17     key usage req: "nonRepudiation, digitalSignature, keyEncipherment"
18     key usage ca: "critical, digitalSignature, cRLSign, keyCertSign"
19     pathlen: "5"
```

Abb 22) Hier werden z.B. die Verwendungsmöglichkeiten des Zertifikats für eine Issue CA definiert.

Der Schritt zur Erstellung des Schlüsselpaars, ist identisch zu der Root-CA-Rolle. Danach wird der zuvor erstellte Schlüssel bei der CSR-Erstellung genutzt (Siehe Abb 23). Das nun erstellte CSR wird in den public Ordner von Nginx abgelegt und kann nun von dort heruntergeladen werden. Der Speicherort für die CSR bzw Nginx-Root-Verzeichnis, wird in dem group\_vars für die Host-Grout "pki" definiert (Siehe Abb 24).

```

roles > intermediate_ca_csr > tasks > ! create_certificate_signing_request.yml

3  - name: ..... Create csr file
4    shell: |
5      ..... openssl req \
6      ..... -new \
7      ..... -sha256 \
8      ..... -nodes \
9      ..... -config {{ pki_opensslconfig }} \
10     ..... -key {{ pki_priv_key }} \
11     ..... -out {{ pki_csr_file }}

```

Abb 23) In Zeile 10, sieht man das der private Schlüssel als Parameter übergeben wird, aber tatsächlich wird nur der öffentliche Schlüssel in das CSR-File eingebettet. Auf magische Weise, generiert OpenSSL aus dem privaten Schlüssel den Öffentlichen.

```

inventories > production > group_vars > ! pki.yml

28
29  pki_workspace: ..... "/srv/pki"
30  pki_publication_dir: ..... "/srv/pki/html"
31  pki_ca_dir: ..... "{{ pki_workspace }}/ca"
32  pki_csr_local_dir: ..... "{{ pki_workspace }}/csr"
33  pki_end_entities_dir: ..... "{{ pki_workspace }}/end_entities"
34

```

Abb 24)

Zum Schluss wird in der Datei `check_files.yml` der Role `intermediate_ca_csr` noch einmal die generierten Dateien auf Plausibilität getestet. Sollte es zu Problemen kommen, soll das die Fehlersuche vereinfachen. Hat die Variable `debug_output` nicht den Wert `"true"` wird diese Prüfung übersprungen

## Das Signieren der Intermediate CA Zertifikats

Wie in Tabelle 1 schon zu sehen ist, gibt es keine extra Ansible-Role in der die Root-CA Zertifikate ausstellt. Der Grund dafür ist, dass sich dieser Prozess nicht von dem der Intermediate CAs unterscheidet. Deswegen wird die Role `intermediate_ca_signing` einfach für den Host `pki_root_ca` verwendet (Siehe Abb 25).



```

! pki_bootstrap.yml
40
41 - name: Signed policy ca cert by the root ca
42   hosts: pki_root_ca
43   roles:
44     - intermediate_ca_signing
45

```

Abb 25)

Es wird Zeit sich anzuschauen was in der Role *intermediate\_ca\_signing* passiert. Zunächst wird das eigene signierte Zertifikat von der nächsthöheren CA über http heruntergeladen. Die Root-CA hat keine CA über sich, deshalb lädt sich das Zertifikat von selbst bei sich selbst herunter. Die Root-CA bräuchte eigentlich ihre Zertifikate gar nicht über http herunterladen, weil sie schon lokal vorhanden sind. Da die Root-CA jedoch ihre Zertifikate über http verteilen muss, wird hier nicht auf eine Condition (Bedingung) zurückgegriffen, sondern die Root-CA wie jede andere CA behandelt.

Anschließend wird das Root-Zertifikat heruntergeladen. Dieser Schritt erfolgt auf jeder CA den gesamten PKI-Baum abwärts. Auf diese Weise wird das Root-Zertifikat auf allen CAs verteilt, so dass es sich Clients dort herunterladen und installieren können. Das ist erforderlich damit die Clients später abgeleitete Zertifikatsketten (Chain of Trust) auf Vertrauenswürdigkeit überprüfen können.

Dieser Schritt ist bei einer Root-CA tatsächlich überflüssig. Deshalb gibt es in Zeile 19 Abb 26 eine Condition (Bedingung), die prüft ob es sich bei dem Host um eine Root-CA handelt. Trifft dies zu, wird der Schritt übersprungen, da das Zertifikat schon exakt an dieser Stelle liegt. Für das anschließende Herunterladen des Zertifikats-Bundle, gibt es noch einmal eine Condition die die Root-CA ausschließt, da es für die Root-CA kein sinnvolles Zertifikats-Bundle geben kann.

```

roles > intermediate_ca_signing > tasks > ! download_trust.yml
15 - name: Download root ca certificate files from the referenc ca
16   get_url:
17     url: "{{ referenc_ca_url }}/{{ pki_root_ca_cert }}"
18     dest: "{{ pki_publication_dir }}/{{ pki_root_ca_cert }}"
19     when: root_ca is not defined
20     ignore_errors: "{{ debug_output }}"
21     register: shell_result
22

```

Abb 26)

In Datei *create\_bundle.yml* geht es weiter mit der Erstellung der Zertifikats-Bundle. Ein Zertifikats-Bundle enthält alle notwendigen Zwischen-Zertifikate (Intermediate-Zertifikate), die notwendig sind, um die Chain-of-Trust lückenlos zurückverfolgen zu können. Mit GUI-Tools wie XCA und Kleopatra kann man das ganz gut visualisieren (siehe Abb 27). Eine Zertifikats-Bundle-Datei ist eigentlich nur eine Textdatei. Wichtig ist aber richtige Reihenfolge der darin zu findenden Zertifikate. Die höchste oder oberste CA, die Root-CA muss als

letztes ins Bundle-File kommen, und das Zertifikat der End-Entity als erstes ganz oben im Bundle-File. Der PKI-Baum steht quasi auf dem Kopf in dem Bundle-File. Programme wie Nginx sind da sehr streng. Bei der falschen Reihenfolge verweigern sie die Zusammenarbeit.

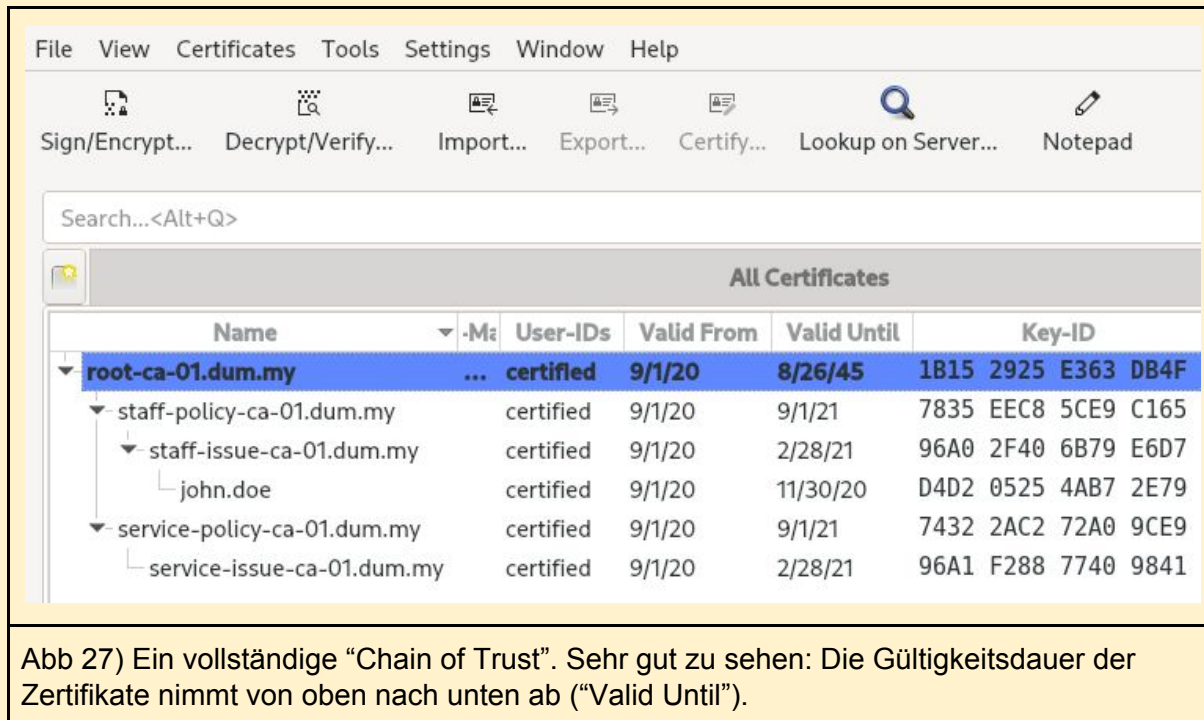


Abb 27) Ein vollständige "Chain of Trust". Sehr gut zu sehen: Die Gültigkeitsdauer der Zertifikate nimmt von oben nach unten ab ("Valid Until").

Bevor jetzt die CA endlich ein Zertifikat beglaubigen kann, muss es noch das CSR herunter laden. Das geschieht in der Datei *download\_csr.yml*. Eigentlich müsste man sagen "hochladen". Die CA holt sich das CSR-File von CAs die in der Hierarchie unter ihr stehen. Der Prozess ist ein Pull- bzw. HTTP-GET-Request.

An dieser Stelle wird es noch mal philosophisch oder paranoid, -je nach persönlicher Sichtweise. Die Frage ist nämlich, wie kann die CA überprüfen, ob der Inhalt in dem CSR stimmt, also ob der Public-Key in dem CSR tatsächlich dem gehört, der namentlich in dem CSR als Common-Name genannt wird? Das ist eine berechnete Frage! In unserem PoC gehen wir von folgenden Annahmen aus:

- auf die Git-Repo, aus der das Ansible stammt, hat nur ein Kreis von berechtigten Personen Zugriff
- die vertrauenswürdigen Commits werden auch daran erkannt, dass sie gültige Signaturen von berechtigten Personen haben
- alle beteiligten Komponenten (Server) werden von dem selben Playbook gesteuert
- die IP-Adressen können nicht manipuliert werden, somit wissen die CAs von wem sie die CSR herunter laden

Die Kommunikation ist zwar nicht verschlüsselt, aber da keine Geheimnisse (wie private Schlüssel) ausgetauscht werden ist das irrelevant. Wenn diese Sicherheit nicht hoch genug ist, wird es aufwendig.



Nun kommt der eigentliche Signatur-Prozess in Datei *signed\_certificate.yml* (Abb 28): In Zeile 6+9 wird wie bei dem Root-CA eine Konfiguration übergeben, um zu verhindern, dass OpenSSL in den Dialog-Modus fällt. In Zeile 10 wird dann auch die CSR-Datei übergeben. Das signierte Zertifikat, wird in Zeile 11 in das Verzeichnis des Nginx zur Veröffentlichung abgelegt. Von hier holt sich der Eigentümer des Zertifikats per http get. Das kann aber auch jeder anders herunterladen, doch ohne privaten Schlüssel nützt das nichts. In der Datei *check\_files.yml* werden die erstellten Dateien nur noch überprüft, damit Fehler möglichst früh auffallen.

```
roles > intermediate_ca_signing > tasks > ! signed_certificate.yml
3  - name: ..... Generate and sign immediate CA certificate
4    shell: |
5      ..... openssl ca \
6      ..... -config {{ pki_opensslconfig }} \
7      ..... -days {{ item.value.not_after_days }} \
8      ..... -notext \
9      ..... -batch \
10     ..... -in {{ pki_csr_local_dir }}/{{ item.key }}.csr.pem \
11     ..... -out {{ pki_publication_dir }}/{{ item.key }}.crt.pem
12   args:
13     chdir: ..... "{{ pki_private_dir }}"
14     creates: ..... "{{ pki_publication_dir }}/{{ item.key }}.crt.pem"
15     with dict: ..... "{{ pki_owner_csr }}"
16     ignore_errors: ..... "{{ debug_output }}"
17     register: ..... shell_result
18
```

Abb 28)

Somit hat jetzt die beantragende CA ihr signiertes Zertifikat und kann damit ihrerseits anderen CAs ihrer Zertifikate auf dieselbe Art signieren. In Abb 29 sieht man wie die CAs der Reihe nach abwärts durch zertifiziert werden, bis ganz unten zu den End-Entities den Services und Benutzern (Clients). Das sind die letzten Glieder der Kette. Wenn diese miteinander kommunizieren (sich gegenseitig ihrer signierten Zertifikate austauschen) ist der Kreis geschlossen, den wir schon ganz zu Anfang in der Abb 2+3 gesehen haben.

```

! pki_bootstrap.yml
36  - name: Creating CSR and self signed root certificate
37    hosts: pki_root_ca
38    roles:
39      - root_ca_self_signing
40
41  - name: Signed policy ca cert by the root ca
42    hosts: pki_root_ca
43    roles:
44      - intermediate_ca_signing
45
46  - name: Signed issue ca cert by the policy ca
47    hosts: pki_policy_ca
48    roles:
49      - intermediate_ca_signing
50
51  - name: Signed end entities cert
52    hosts: pki_issue_ca
53    roles:
54      - intermediate_ca_signing
55
56  - name: Create fake services for tests
57    hosts: pki_fake_service
58    roles:
59      - fake_service
60
61  - name: Create a fake client for tests
62    hosts: pki_fake_client
63    roles:
64      - fake_client
65

```

Abb 29)

## Die Ansible-Roles der End-Entities

Da die End-Entities in diesem PoC ihre CSRs und Zertifikate genauso austauschen wie die CAs, brauchen diese auch einen Nginx. Für eine End-Entity die ein Web-Service ist, stellt das keine Hürde dar. Anders auf der Gegenseite: wenn das Auszustellende CSR und Zertifikat zu einem Mitarbeiter(-End-Entity) gehört, wird dieser nicht standardmäßig einen Nginx auf seinem Notebook haben und das Notebook wird auch nicht immer eine bekannte

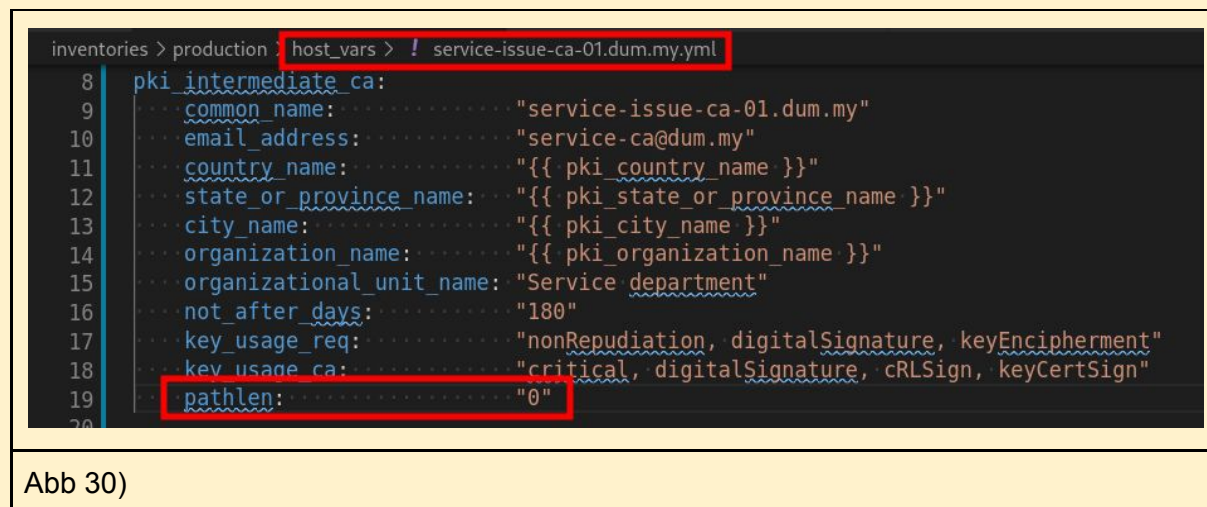
IP-Adresse besitze, häufig wird das Notebook nicht mal immer im selben Netzwerk sein. Diese Problematik wird in dem PoC nicht berücksichtigt.

Das Problem der Client-Zertifikate ist technisch nicht leicht zu lösen, denn auf irgendeine Weise muss die Issue-CA das das Zertifikat der End-Entity signieren soll, prüfen können, dass das CSR wirklich von dem Richtigem stammt und nicht manipuliert wurde. Ein Lösungsansatz wäre, der Mitarbeiter erstellt sein CSR selber und lädt es auf ein Verzeichnis (per ssh oder Network-Share z.B.), auf das nur er Zugriff hat. Dieses Verzeichnis, wird da per http für die Issue-CA zugänglich gemacht. Dann bleibt die Frage zu klären, ob und wie man diesen Teil automatisiert.

Die Ansible-Roles die zu der PKI-Komponente End-Entity gehören sind:

- *end\_entity\_csr*
- *fake\_client*
- *fake\_service*

Es gibt keine Ansible-Role, die speziell das Signieren von End-Entities abbildet. Dazu wird die Ansible-Role *intermediate\_ca\_signing* verwendet, die in den *host\_vars*-Files nur leicht verändert parametrisiert wird (Abb 30).



Ein weiterer wichtiger Unterschied zwischen CA- und End-Entity-Zertifikat bzw CSR ist in der Konfiguration (`roles/end_entity_csr/templates/openssl.cnf`) ist in Zeile 90 und 96 zusehen (Abb 31). Die End-Entities haben keine Bevollmächtigung ihrerseits Zertifikate zu signieren. Sie sind eben *KEINE* CAs! Es liegt in der Verantwortung der CAs nur die CSR zu akzeptieren, die die korrekten Berechtigungen beantragen. In diesem PoC wird nicht davon ausgegangen, dass die Teilnehmer der PKI bösartig sind. Die Identitäten sind bekannt und eine boshafte Handlung ließe sich zwar nicht verhindern, aber zurückverfolgen. Wird diese Grundannahme verneint, wird es sehr schnell sehr aufwendig, geeignete Maßnahmen zu ergreifen, die diese Sabotage verhindern.

```

roles > end_entity_csr > templates > openssl.cnf
90  basicConstraints ..... = CA:FALSE
91  nsComment ..... = "OpenSSL Generated Certificate"
92  subjectKeyIdentifier ..... = hash
93  authorityKeyIdentifier ..... = keyid, issuer
94
95  [ v3_req ]
96  basicConstraints ..... = CA:FALSE
97  keyUsage ..... = {{ end_entity.key_usage_req }}
98

```

Abb 31)

## Die Service End-Entity

VMs die in unserem PoC die PKI-Komponente eines Service End-Entity repräsentiert, werden über die Rollen *end\_entity\_csr* und *fake\_service* abgebildet. Zu der ersten Role ist bereits alles wichtige gesagt: Sie erstellt eine CSR-Datei und hält diese über ein Nginx für die Issue-CA bereit zum Download.

In der Role *fake\_service* wird zuerst, wie schon in der Role *intermediate\_ca\_signing*, die signierten Zertifikate von der Issue-CA heruntergeladen und das Bündel-File um das eigene Zertifikat erweitert. Dann werden verschiedene Verzeichnisse und index.html-files erstellt, die Services mit verschiedenen Sicherheitsstufen repräsentieren, die durch eine Mutual-Authentication geschützt werden sollen.

```

roles > fake_service > templates > ⚙ fake.conf
6
7  server {
8      listen .....{{ service_port }} ssl http2;
9      server_name .....{{ end_entity.common_name }};
10
11     ssl .....on;
12     ssl_certificate .....{{ pki_bundle_file }};
13     ssl_certificate_key .....{{ pki_priv_key }};
14     ssl_protocols .....TLSv1.2;
15
16     ssl_verify_client .....optional;
17     ssl_client_certificate .....{{ pki_bundle_file }};
18     ssl_verify_depth .....5;
19
20     root .....{{ pki_service_http_root_dir }};
21     error_log ...../srv/http/debug.log debug;
22
23     location ...../oneway/ {
24         autoindex .....on;
25     }
26
27     location ...../mutual/ {
28         if .....($ssl_client_verify != SUCCESS) {
29             return .....403;
30         }
31         autoindex .....on;
32     }
33
34     location ...../company-mutual/ {
35         if .....($ssl_client_s_dn !~ "OU=Fooobar staff department") {
36             return .....403;
37         }
38         autoindex .....on;
39     }
40 }

```

Abb 32)

Der interessante Teil passiert in der Konfiguration der Nginx-Konfiguration, die über das Template-System von Ansible generiert wird. In Abb 32 ist das Template zu sehen. Zeile 11 bis 13 werden schon die meisten gesehen haben. Hier werden der private Schlüssel und das Bundle mit dem eigenen Zertifikat eingebunden. Dies wird benötigt, damit der Client eine Verbindung per HTTPS aufbauen kann. In Zeile 14 werden die akzeptierten Protokoll-Versionen eingeschränkt. In Zeile 16 wird die Client-Verification eingeschaltet. Hier wird sie auf "optional" gesetzt, denn in Zeile 23 bieten wir auch Inhalte ohne Client-Verification an. Haben wir Inhalte die ausschließlich für Client-Verification vorgesehen sind, sollte statt "optional" besser "on" gesetzt werden.

In Zeile 17 (*ssl\_client\_certificate*) wird ein Bundle von allen CA-Zertifikaten übergeben, denen als Issue-CA vertraut wird. Das Client-Zertifikat muss von einer dieser CAs unterzeichnet worden sein, um akzeptiert zu werden. Da in einer PKI über eine Trust-of-Chain Aufgaben delegiert werden können, wird noch in Zeile 18 (*ssl\_verify\_depth*) eingeschränkt, wie lang die Trust-of-Chain maximal sein darf.



In Zeile 23 ist ein Verzeichnis definiert, für das keine Client-Certificate-Verification notwendig ist. In Zeile 27 wird definiert, dass auf das Verzeichnis `/mutual/` jeder zugreifen darf, der über ein gültiges Zertifikat verfügt. In Zeile 34 wird bestimmt, dass auf das Verzeichnis `/company-mutual/` nur zugreifen darf, wer ein Zertifikat besitzt, das auf die Abteilung (engl. Organisation Unit) `"Foobar staff department"` ausgestellt wurde. `ssl_client_s_dn` steht für "subject DN" also das Zertifikat zum Client. Man kann aber auch die Bedingungen an Ausstellende CA knüpfen. Dazu muss man dann die Werte der `ssl_client_i_dn` untersuchen. Das "i" steht hier also für "Issue CA".

## Die Client End-Entity

Um zu testen ob Service End-Entities wirklich nur Requests mit gültigen Zertifikaten akzeptieren, gibt die Ansible-Rolle `"fake_client"` die die Aufrufe eines Mitarbeiters mit dem Browser simulieren.

Zunächst wird in `roles/fake_client/tasks/download_trust.yml` wieder Zertifikat- und Bundle-File heruntergeladen. Da es sich um die Simulation einer Mitarbeiter-PC handelt soll das Root-CA der PKI systemweit bekannt gemacht werden (`roles/fake_client/tasks/sys_import_trust.yml`). Das Handhaben ist je nach Linux-Distribution z.T. unterschiedlich. Dafür brach es dann immer eine Sonderlocke. Die Sonderbehandlungen sind immer am `"when: ansible_distribution != 'XXX'"` im Code zu erkennen.

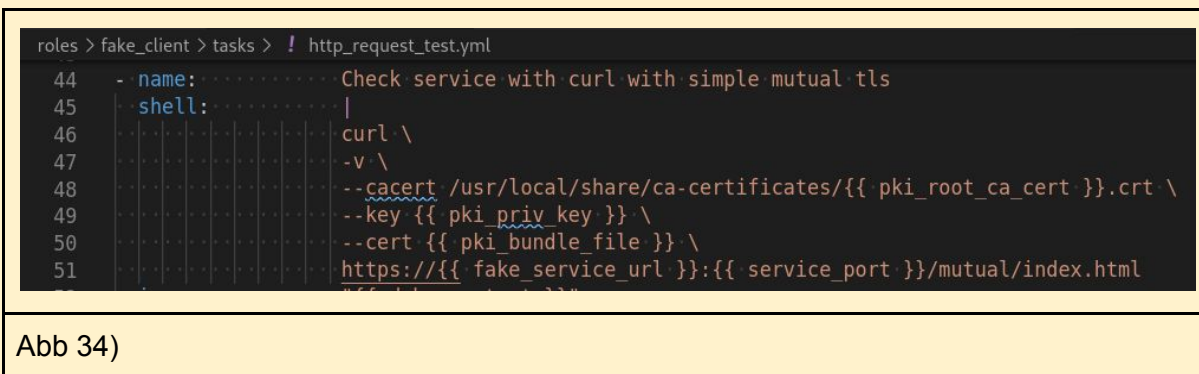
Als Nächstes wird ein Bundle-File mit dem Client-Zertifikat erstellt. Der Client muss die gesamte Zertifikatskette übergeben und zwar soweit, bis eine CA erreicht ist, die auch der Service kennt und der der Service auch vertraut. Wenn wir uns noch mal die Abb. 2+3 ansehen, stellen wir fest, dass der Service in einem völlig anderen Zweig der PKI steckt als der Client. Also muss der Client die komplette Kette, bis hinauf zur Root-CA mitliefern.

```
roles > fake_client > tasks > ! http_request_test.yml
20
27 - name: ..... Check service with curl with explicit cacert
28   shell: .....|
29   ..... curl \
30   ..... -v \
31   ..... --cacert /usr/local/share/ca-certificates/{{ pki_root_ca_cert }}.crt \
32   ..... https://{{ fake_service_url }}:{{ service_port }}/oneway/index.html
```

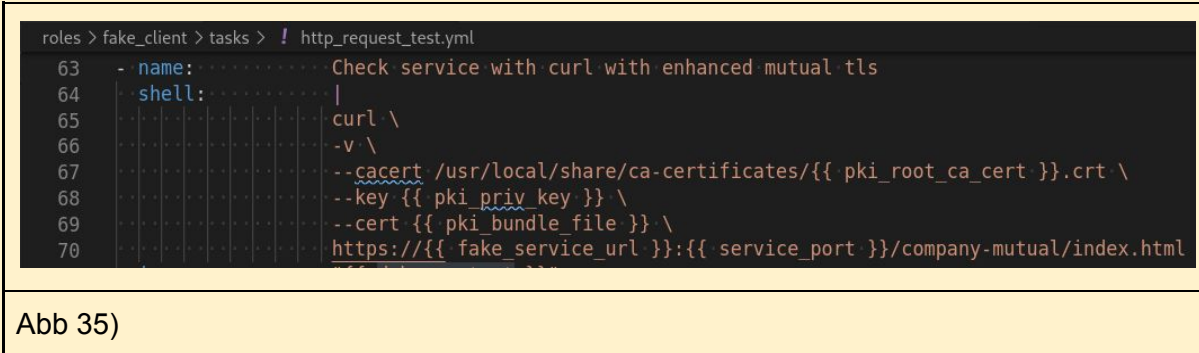
Abb 33)

In `roles/fake_client/tasks/http_request_test.yml` erfolgen die eigentlichen Request-Tests per curl-Aufruf. Da der PoC kein DNS hat, wird die Domain mit dem Test-Service in `/etc/hosts` eingetragen. In Zeile 29-32, Abb. 33, wird getestet, ob ein Service-Seitiges TLS funktioniert. Das Root-CA-Zertifikat wird hier explizit mitgegeben, um sicherzustellen, dass es nicht zu Fehlern kommt, die distributions-bedingt sind, aber nichts mit der Kernfunktionalität zu tun haben.





In Abb 34 wird der Test erweitert auf Mutual-TLS. Dazu wird eine andere Ressource (/mutual/index.html) aufgerufen und zusätzlich in Zeile 49 der private Schlüssel verwendet und in Zeile 50 das Zertifikats-Bundle mitgegeben.



Der letzte Test unterscheidet sich nur in Resource-Pfade (Abb 35, Zeile 70). Hier überprüft die Gegenseite, ob der Client ein Zertifikat hat, das auf die richtige Abteilung ausgestellt wurde.

# End-Entity Zertifikate im p12-Format für die Verwendung im Browser

Wie im Beispiel oben zu sehen, kommt das Tool curl mit den erstellten Zertifikaten sehr gut zurecht. Andere Browser wie Firefox, Chromium und Chrome wollen ein anderes Format *PKCS#12* oder auch *PFX*. In einer solchen Datei ist das Client-Zertifikat, der private Schlüssel und die CA-Chain enthalten.

```

roles > fake_client > tasks > ! create_p12_file.yml
3  - name:           Crwate a p12 file for browsers
4    shell:          |
5                    openssl pkcs12 \
6                    -export \
7                    -out {{ pki_p12_file }} \
8                    -in {{ pki_bundle_file }} \
9                    -inkey {{ pki_priv_key }} \
10                   -passout pass:
11  args:
12    creates:        "{{ pki_p12_file }}"
13    ignore_errors:  "{{ debug_output }}"
14    register:       shell_result

```

Abb 41)

Dieses p12-Dateiformat wird auch vom Ansible-Playbook auch erstellt, und zwar in der Datei roles/fake\_client/tasks/create\_p12\_file.yml (Abb 41). Im Playbook wird diese Zertifikatsformat auch mit curl getestet. Wenn die Variable *debug\_output* auf "true" gesetzt wird (Siehe Abb 37, Zeile 8), wird auch das generierte p12-Zertifikat für den Browser vom Ansible-Playbook heruntergeladen (siehe Abb 42). Es liegt danach in dem selben Verzeichnis, in dem das Ansible-Playbook ausgeführt wurde.

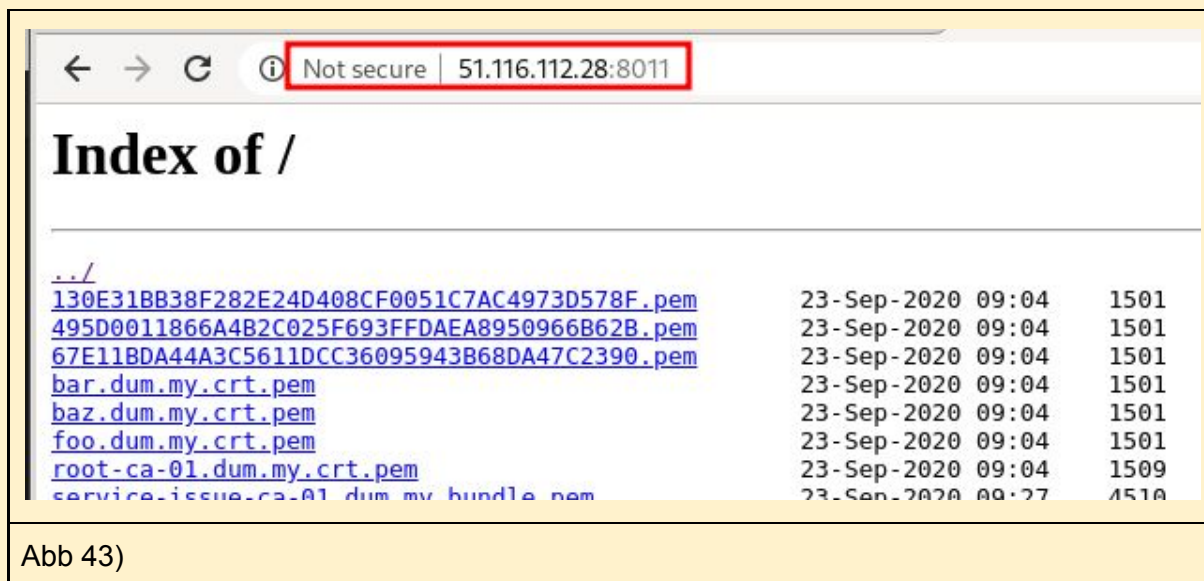
```

roles > fake_client > tasks > ! create_p12_file.yml
60  - name:           Storing p12-certifcat on ansible host for test tasks
61    fetch:
62      src:           "{{ pki_p12_file }}"
63      dest:          ../{{ inventory_hostname }}.p12
64      when:          debug_output

```

Abb 42)

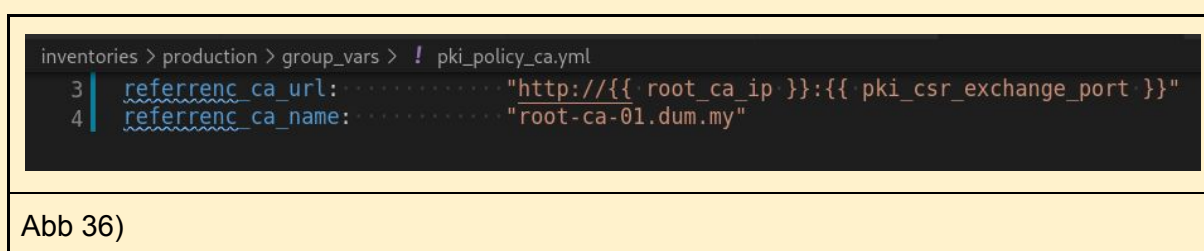
Diese Datei muss noch in dem Browser importiert werden. Damit der Browser das Fake-Service-Zertifikat akzeptiert, muss das Root-CA-Zertifikat heruntergeladen werden. Dazu zu ruft man den Server, auf dem der Fake-Service läuft, über Port 8011 auf. Das ist das Verzeichnis, in dem die Dateien liegen, die die CAs untereinander austauschen. Nun liegt hier auch das Root-CA-Zertifikat bereit (Abb 43). Zuletzt muss noch ein Eintrag in der /etc/hosts gemacht werden, der die IP-Adresse des Fake-Services auf die URL *foo.dum.my* auflöst. Nun kann der Fake-Service im Browser unter der URL ***https://foo.dum.my:8443/company-mutual/index.html*** aufgerufen werden.



## Inventory

Einige grundsätzliche Aspekte sind bereits oben abgehandelt worden (siehe Abb 4+5). Hier geht es noch mal um den Aufbau der *host\_vars* im Detail. Über die *host\_vars*-Files wird die PKI-Architektur gestaltet. Werden mehr oder weniger Hierarchiestufen in dem PKI-Baum benötigt oder werden weitere Nebenlinien gebraucht, so wird das über diese Daten gesteuert.

Es gibt nur sehr wenige Eigenschaften, die sich eine Gruppe von CAs in einer Hierarchieebene teilen, deshalb sind die Konfigurationsdateien in der *inventories/production/group\_vars* sehr leer. Es teilen sich gerade mal Policy CAs gemeinsam ihre Issue CA (Abb 36)



Die Konfiguration der Root-CA in Abb 37 ist da schon wesentlich umfangreicher. Grob lassen sich die Konfiguration in drei Bereiche einteilen: Der erste Teil liefert Informationen zu der Issue-CA bzw "referrenc\_ca" (Zeile 5+6), also der Instanz, die die Zertifikate für die CA ausstellt, von der die Konfiguration ist. In diesem Fall handelt es sich um die Root-CA. Somit verweist sie auf sich selbst. Bei einer Intermediate-CA steht dort die CA die in der Hierarchie der PKI über ihr ist. Allerdings nicht in Abb 38, weil es sich um eine Gruppe von CAs handelt, die eine gemeinsame Issue-CA (referrenc\_ca) haben. Deshalb steht sie hier in der Datei, die in Abb 36 zusehen ist.

```

inventories > production > host_vars > ! root-ca-01.dum.my.yml
3  ansible_host: ..... "{{ root_ca_ip }}"
4
5  referrenc_ca_url: ..... "http://{{ root_ca_ip }}:{{ pki_csr_exchange_port }}"
6  referrenc_ca_name: ..... "root-ca-01.dum.my"
7
8  root_ca: ..... true
9  pki_root_ca:
10     common_name: ..... "root-ca-01.dum.my"
11     email_address: ..... "ca-admin@dum.my"
12     country_name: ..... "DE"
13     state_or_province_name: ..... "Dummyfild"
14     city_name: ..... "Dummy Town"
15     organization_name: ..... "Dummy GmbH"
16     organizational_unit_name: ..... "Security department"
17     not_after_days: ..... "9125"
18     key_usage_req: ..... "nonRepudiation, digitalSignature, keyEncipherment"
19     key_usage_ca: ..... "critical, digitalSignature, cRLSign, keyCertSign"
20
21  pki_owner_csr:
22     "staff-policy-ca-01.dum.my":
23         owner_url: ..... "http://{{ policy_ca_staff_ip }}:{{ pki_csr_exchange_port }}"
24         not_after_days: ..... "365"
25     "service-policy-ca-01.dum.my":
26         owner_url: ..... "http://{{ policy_ca_service_ip }}:{{ pki_csr_exchange_port }}"
27         not_after_days: ..... "365"

```

Abb 37)

Der mittlere Abschnitt einer CA-Host-Konfigurationsdatei, beschreibt die Eigenschaften von sich selbst. Der Block beginnt mit "pki\_intermediate\_ca:" (Abb 38 Zeile 5). Handelt es sich um die Root-CA, beginnt der Block mit "pki\_root\_ca:" (Abb 37 Zeile 9) und die Variable "root\_ca" ist auf "true" gesetzt (zeile 8). In diesem Abschnitt wird festgelegt, welche Attribute die CA hat, wie lange ihr Zertifikat gültig ist und wofür das Zertifikat verwendet werden darf (Zeile 10-19 in Abb 37 und Zeile 6-16 in Abb 38)

Der dritte und letzte Abschnitt beginnt mit "pki\_owner\_csr:" und enthält eine Liste der von CSRs mit ihren Speicherorten, die die CA zu suchen und zu signieren hat (Abb 37 ab Zeile 21 und Abb 38 ab Zeile 20). In der PoC ist es so, das die CA die Attribute der CSR nicht noch mal überprüft. Wollte man das tun, müsste die Liste in diesem Abschnitt viel mehr Informationen zu den CSRs haben.



```

inventories > production > host_vars > ! service-policy-ca-01.dum.my.yml
3  ansible_host: ..... "{{ policy_ca_service_ip }}"
4
5  pki_intermediate_ca:
6    common_name: ..... "service-policy-ca-01.dum.my"
7    email_address: ..... "service-ca@dum.my"
8    country_name: ..... "{{ pki_country_name }}"
9    state_or_province_name: ..... "{{ pki_state_or_province_name }}"
10   city_name: ..... "{{ pki_city_name }}"
11   organization_name: ..... "{{ pki_organization_name }}"
12   organizational_unit_name: "Security department"
13   not_after_days: ..... "365"
14   key_usage_req: ..... "nonRepudiation, digitalSignature, keyEncipherment"
15   key_usage_ca: ..... "critical, digitalSignature, cRLSign, keyCertSign"
16   pathlen: ..... "0"
17
18
19
20  pki_owner_csr:
21    "service-issue-ca-01.dum.my":
22      owner_url: ..... "http://{{ issue_ca_service_ip }}:{{ pki_csr_exchange_port }}"
23      not_after_days: ..... "180"
24    ### EXAMPLE
25    # "service-issue-ca-02.dum.my":
26    #   owner_url: ..... "http://{{ issue_ca_service_ip }}:{{ pki_csr_exchange_port }}"
27    #   not_after_days: ..... "180"

```

Abb 38)

In Abb 39 ist die Konfiguration einer Servic-End-Entity zusehn. Sie besteht nur aus zwei Blöcken. Dem ersten Block, mit den Informationen zur Issue-CA (referrenc\_ca) und den zweiten Block der mit "end\_entity:" beginnt und sich selbst beschreibt. Ein Block "pki\_owner\_csr:" fehlt hier, weil es keine CA ist die CSRs bearbeitet.

```

inventories > production > host_vars > ! foo.dum.my.yml
3  ansible_host: ..... "{{ foo_dum_my_ip }}"
4
5  referrenc_ca_url: ..... "http://{{ issue_ca_service_ip }}:{{ pki_csr_exchange_port }}"
6  referrenc_ca_name: ..... "service-issue-ca-01.dum.my"
7
8  end_entity:
9    common_name: ..... "foo.dum.my"
10   email_address: ..... "service-ca@dum.my"
11   country_name: ..... "{{ pki_country_name }}"
12   state_or_province_name: ..... "{{ pki_state_or_province_name }}"
13   city_name: ..... "{{ pki_city_name }}"
14   organization_name: ..... "{{ pki_organization_name }}"
15   organizational_unit_name: "Foobar service department"
16   not_after_days: ..... "90"
17   key_usage_req: ..... "nonRepudiation, digitalSignature, keyEncipherment"
18   key_usage_ca: ..... "critical, digitalSignature, cRLSign, keyCertSign"
19

```

Abb 39)

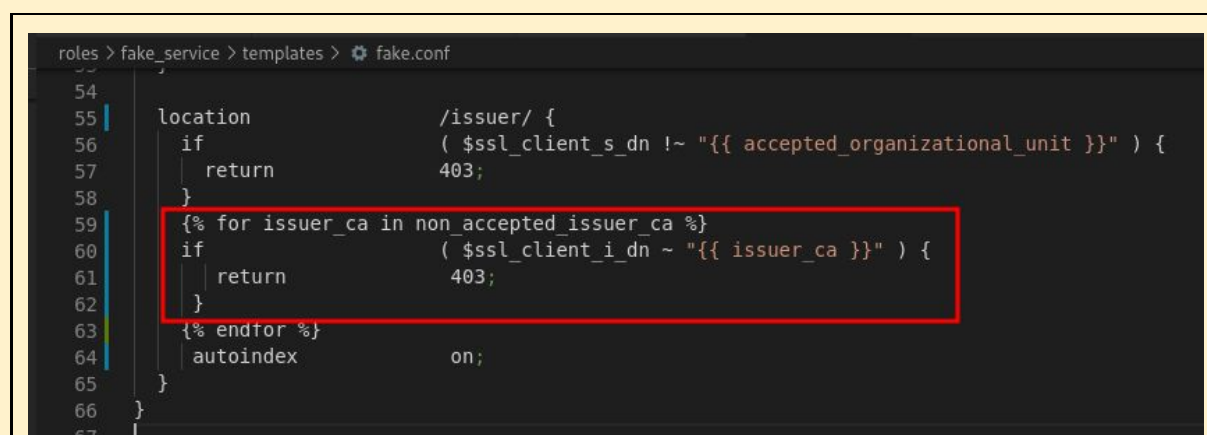
# Certificate Revocation List (CRL)

Die Certificate Revocation List (CRL, zu Deutsch: Zertifikatsperrliste) dient dazu, zertifikate zu widerrufen bevor sie abgelaufen sind. Dies wird gemacht, wenn die Zertifikate fehlerhaft ausgestellt wurden, oder die CA kompromittiert wurde.

Die Erstellung, Verteilung und Verwaltung von CRLs benötigt eine Infrastruktur die vergleichbar aufwendig ist wie die PKI selbst. Der Mehrwert dafür ist aber nur gering denn es gibt eine Reihe von Problemen mit CRLs die sich architektur bedingt nicht lösen lassen.

- Die Teilnehmer der PKI sind nicht gezwungen, die CRLs zu berücksichtigen
- Nicht jeder PKI-Teilnehmer unterstützt CRLs
- CRLs können veraltet sein
- CRLs könnten nicht erreichbar sein
- Bei mehreren CAs gibt es mehrere CRLs zu verwalten
- CRLs können mit der Zeit sehr umfangreich werden
- Es können keine Zertifikate widerrufen werden, von denen man nicht weiss, das sie erstellt wurden (bei kompromittiert Systemen)

Als Alternative zu CRLs kann man stattdessen, alle Zertifikate verwerfen, die von einer bestimmten Issuer CA erstellt wurde, von der man weiss, das sie kompromittiert wurde. Das ist eine sehr rabiante methode, hätte aber die Vorteile, dass man keine komplexe Infrastruktur verwalten muss. Dadurch würden vielleicht viel mehr Zertifikate für ungültig erklärt werden als notwendig, aber da sowieso das erklärte ziel ist, Zertifikate maximal schnell und automatisch austauschen zu können, sollten wir mit dem Nachteil gut leben können.

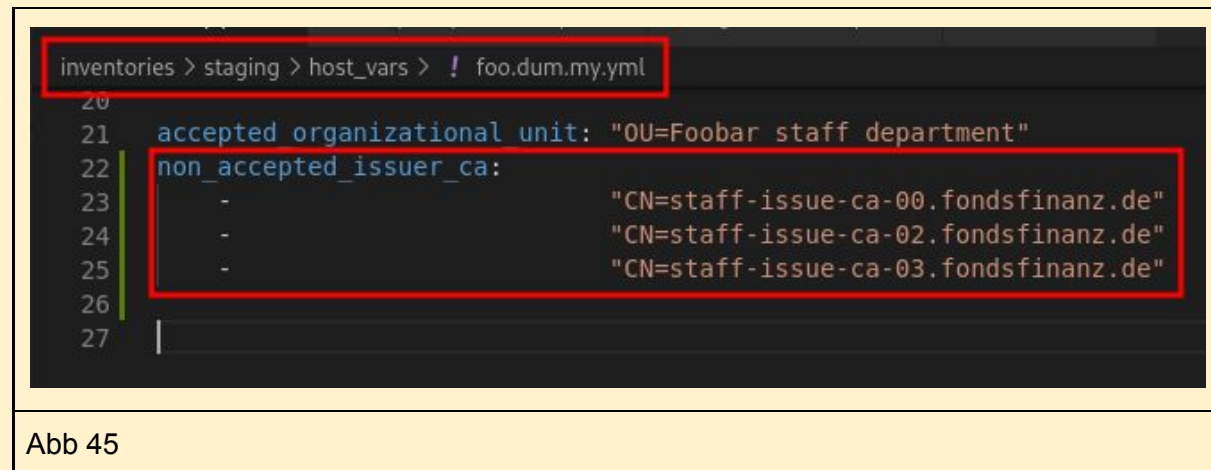


```
roles > fake_service > templates > fake.conf
54
55 location /issuer/ {
56     if ( $ssl_client_s_dn !~ "{{ accepted_organizational_unit }}" ) {
57         return 403;
58     }
59     {% for issuer_ca in non_accepted_issuer_ca %}
60     if ( $ssl_client_i_dn ~ "{{ issuer_ca }}" ) {
61         return 403;
62     }
63     {% endfor %}
64     autoindex on;
65 }
66 }
```

Abb 44

In Abb 44 ist zusehen, wie eine Regel in Nginx realisiert wird. Hier wird in einer Schleife generisch mehrere Regeln generiert um Issuer CAs abzugleichen. Diese Regeln müssen

nur solange vorgehalten werden, bis das Zertifikat der betreffenden CA sowieso abgelaufen ist.



In dem Inventory-File des Service werden die Issue CAs als Liste hinterlegt, die verworfen werden sollen (Abb 45). Hier sieht man das die CAs immer neue Namen Bekommen (hoch nummeriert), wenn sie von der höheren Instanz zertifiziert werden. Würde man immer den gleichen Namen wählen, würden auch die erneuerte CA abgelehnt werden.

Nginx kann nur das Startdatum des Zertifikats der End Entity prüfen, aber nicht, das der Issuer CA. Wenn die Issuer CA kompromittiert wurde, ist es denkbar, das die Zertifikate der End Entity vordatiert wurden wurden. Deshalb kann man darauf nicht sicher testen.

## Debugging-Modus

Ein Bock, wie in Zeile 16-23 in Abb 21, findet sich an vielen Stellen des Ansible-Playbooks. Wenn die Variable *debug\_output* in der Datei *inventories/production/group\_vars/pki.yml* auf *"true"* gesetzt ist (siehe Abb 4 Zeile 3), wird nicht gleich bei dem Fehler gestoppt sondern noch eine umfangreiche Ausgabe generiert.



```

roles > intermediate_ca_signing > tasks > ! signed_certificate.yml
1  ---
2
3  - name: ..... Generate and sign immediate CA certificate
4    shell: |
5      ..... openssl ca \
6      ..... -config {{ pki_opensslconfig }} \
7      ..... -days {{ item.value.not_after_days }} \
8      ..... -notext \
9      ..... -batch \
10     ..... -in {{ pki_csr_local_dir }}/{{ item.key }}.csr.pem \
11     ..... -out {{ pki_publication_dir }}/{{ item.key }}.crt.pem
12   args:
13     chdir: ..... "{{ pki_private_dir }}"
14     creates: ..... "{{ pki_publication_dir }}/{{ item.key }}.crt.pem"
15     with dict: ..... "{{ pki_owner_csr }}"
16     ignore_errors: ..... "{{ debug_output }}"
17     register: ..... shell_result
18
19   - name: ..... Output return value
20     debug:
21       msg: ..... "{{ shell_result.results }}"
22     when: ..... debug_output
23

```