



Infrastruktur- und Security- Tests

whoami

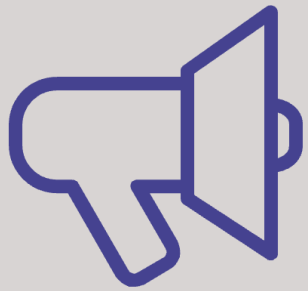


Christian Stankowic

- IT-Berater und -Trainer bei [SVA](#)
 - SUSE, Red Hat, Debian, etc.
 - Patch-Management
 - IaC, Automation, DevO(o)ps
 - Terraform, Packer, Vagrant,...
-  [cstan.io](#)
-  [Faxinformatiker](#), [ThinkPad-Museum](#)

Agenda

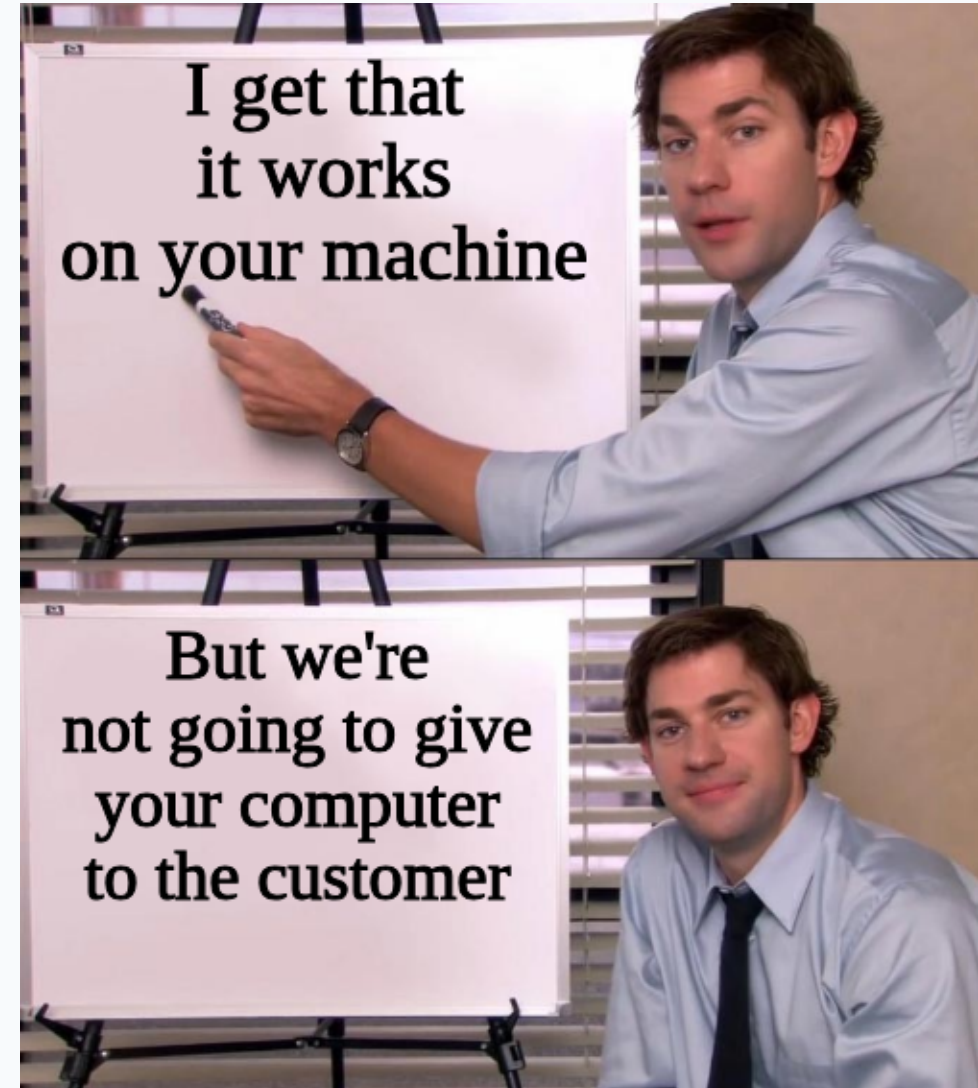
- 1** Motivation
- 2** Tool-Auswahl
- 3** Lessons Learned



Motivation

Motivation

Warum überhaupt Infrastruktur testen,
wenn Entwickler:innen ihren Code
testen? 🤔



Beispiel 1: Lab-Umgebungen

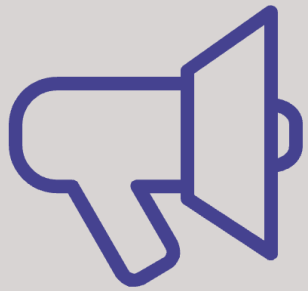
- Trainings benötigen **funktionale** Umgebungen
 - i.d.R. komplexe Setups aus mehreren Hosts, Services und Abhängigkeiten
- Teilnehmende wollen sofort ohne Debugging loslegen können
- Manuelle Eingriffe während eines Workshops unterbrechen den Flow
 - andere Teilnehmende müssen warten
- Lernumgebungen sollten vorab **getestet** werden
 - Sind alle Hosts erreichbar?
 - Sind die Services korrekt konfiguriert?
 - Ist externer Zugriff (Firewall, DNS) wie benötigt möglich?

Beispiel 2: Projekt-Umgebungen

- Umgebungen pro Kund:in unterschiedlich, auch bei gleicher Software
 - Bare-Metal, VMs, Cloud
 - Linux-Distribution und benötigte Services
 - Netzwerk-Topologie
- **Unit-Tests** können dabei helfen, Probleme schnell zu entdecken
 - Airgapped-Umgebung (fehlende Pakete und Artefakte)
 - Es ist **immer** DNS, die Firewall und/oder SELinux
- **Prerequisite**-Tests können Anforderungen vorab verifizieren
 - CPU, Arbeitsspeicher
 - Betriebssystem und Abhängigkeiten
 - Netzwerk

Beispiel 3: Security-Hardening

- Für einige Systeme gibt es bereits automatisierte Tools:
 - [DevSec Hardening Framework](#) für Linux, SSH, Apache, etc.
 - [ansible-lockdown](#) für CIS- und STIG-Hardening
 - [kube-bench](#) für Kubernetes
- Hersteller bieten oft testbare Hardening Guidelines an:
 - [HashiCorp Vault Production Hardening Guide](#)
 - [OWASP Cheat Sheet Series](#)
 - [RHEL 9 Security Hardening Guide](#)
 - [SLES 15 Security and Hardening Guide](#)
 - ...



Tool-Auswahl

Ansible: `--check` und `--diff`

- Die Modi können einzelnen oder zusammen benutzt werden
 - können auf **Task**- oder CLI-Ebene gesetzt werden
 - unterstützen dabei Änderungen abzuschätzen
- `--check` überprüft lediglich, welche Änderungen getätigt werden würden
 - nicht jedes Modul unterstützt dies!
 - aufeinander aufbauende Aktionen werden fehlschlagen
- `--diff` listet auszuführende oder ausgeführte Änderungen auf
 - wird ebenfalls nicht von jedem Modul unterstützt

Ansible: `--check` und `--diff`

Beispielhaftes Playbook:

```
- name: Update configuration
  ansible.builtin.copy:
    dest: /etc/myapp.conf
    content: 'ENABLE=mymod,ssl'
```

Es werden keine Änderungen vorgenommen:

```
$ ansible-playbook --check --diff myapp.yml
...
TASK [Update configuration] *****
--- before
+++ after: /etc/myapp.conf
@@ -0,0 +1 @@
+ENABLE=mymod,ssl
\ No newline at the end of file

$ cat /etc/myapp.conf
cat: /etc/myapp.conf: No such file or directory
```

ansible.builtin.assert

- Teil von `ansible-core`, in jeder Ansible-Installation enthalten
- Überprüft, ob Ausdrücke zutreffen
 - `success` - und `error` -Meldungen
- Nützlich im Zusammenhang mit **Jinja2** und **Ansible Facts**
- Übliche Use-Cases
 - Verifizieren, dass Zielzustand erreicht wurde
 - Prerequisite-Checks vor Installationen
 - Einfaches Testen von Ansible-Rollen
- Nicht sehr handlich für komplexe Infrastruktur-Tests
 - Syntax, Jinja2-/Ansible-Logik

ansible.builtin.assert

Prerequisite-Check:

```
- name: Ensure having supported SP release for SUMA 4.3
  ansible.builtin.assert:
    that:
      - uyuni_suma_release == 4.3
      - ansible_distribution_verison == '15.4'
    success_msg: 'Supported SLES SP found'
    fail_msg: 'Please upgrade to SLES 15 SP4'
```

Log überprüfen:

```
- name: Get application logs
  ansible.builtin.command: crapp --get-logs
  register: myapp

- name: Check if app works properly
  ansible.builtin.assert:
    that:
      - myapp | regex_findall('DEPLOY_OK') | length > 1
    fail_msg: 'Deployment hasn't succeeded, yet!'
    success_msg: 'Application works properly'
```

goss

- Imperative und deklarativ
- Lokale Ausführung
- für Linux, Windows und macOS
- Varianten für Container ([dgoss](#)), docker - compose ([dcgoss](#)) und Kubernetes ([kgoss](#))
- Unterstützt zahlreiche Ressourcen
 - Pakete, Prozesse, Dateien, Kernel-Parameter, Ports, Mounts, etc.

goss

```
file:
  test.txt:
    exists: true
    contents: |
      test file
      second line
```

```
command:
  echo '15':
    exit-status: 0
    stdout:
      and:
        - gt: 10
        - lt: 50
        - match-regexp: '\d{2}'
    timeout: 10000
```

```
http:
  https://ifconfig.me:
    status: 200
    timeout: 5000
    body:
      - '{{.Vars.Ip}}'
```

```
package:
  nginx:
    installed: true
    versions:
      - 1.33.7
```

goss

```
$ goss validate  
....FS..
```

Failures/Skipped:

```
Package: nginx: installed:  
Expected  
    false  
to equal  
    true
```

```
Package: nginx: version: skipped
```

```
Total Duration: 0.223s
```

```
Count: 8, Failed: 1, Skipped: 1
```

serverspec

- [RSpec](#) -Plugin für Infrastruktur-Tests
 - Ruby-basiertes Testing-Framework
- Deklarative Tests
- Ausführung lokal oder via `ssh`, WinRM oder Docker API
- für Linux, Windows und macOS
- über 40 unterstützte Ressourcen
 - Dateien, Schnittstellen, User/Group, Pakete, Cronjobs, Firewall, Ports, etc.
 - siehe auch [Dokumentation](#)

serverspec

```
describe 'localhost' do

  describe file('/etc/passwd') do
    it { should be_file }
  end

  describe command('ls /foo') do
    its(:stderr) { should match /No such file or directory/ }
  end

  describe cron do
    it { should have_entry '* * * * * /usr/local/bin/foo' }
  end

  describe default_gateway do
    its(:ipaddress) { should eq '192.168.10.1' }
    its(:interface) { should eq 'br0' }
  end

  describe package('httpd') do
    it { should be_installed }
  end

end
```

Dev-Sec und Ansible-Lockdown

- Nicht nur das Härten neuer Systeme ist wichtig
- Bestehende Systeme müssen **regelmäßig** erneut auditiert/gehärtet werden
 - hoher Aufwand je nach Größe der Systemlandschaft
- [Dev-Sec](#) auditiert **und** härtet Server
 - Baselines für Linux, SSH, Apache,...
 - Ansible- und Puppet-Automatismen für **Härtung**
- Baselines auf Basis von
 - CIS-Benchmarks
 - NSA Hardening Guides
 - Hersteller Best Practices

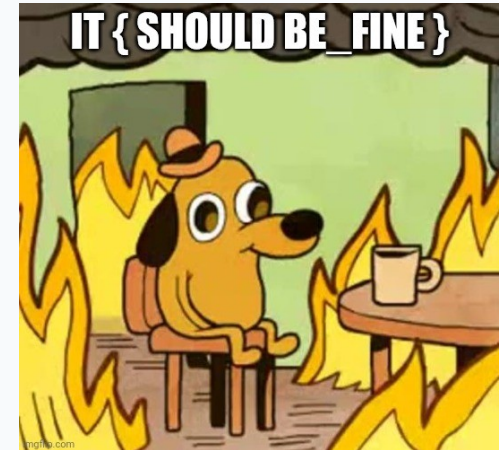
Dev-Sec und Ansible-Lockdown

- Überprüfung via [InSpec](#)
 - Testing-Framework auf [RSpec](#)-Basis, Fokus auf Security
- Alternative: [ansible-lockdown](#)
 - ähnlich Dev-Sec, aber nutzt `goss` für Testing
 - CIS-/STIG-Benchmarks und -Härtung

Auditierung mit Dev-Sec

Beispielhafter Test:

```
control 'package-02' do
  impact 1.0
  title 'Do not install Telnet server'
  desc 'Telnet protocol uses unencrypted communication'
  describe package('telnetd') do
    it { should_not be_installed }
  end
end
```



Kudos an [DDKFM](#)

Auditieren eines Hosts:

```
$ inspec exec linux-baseline -t ssh://user@host
...
[CHECK] package-02: Do not install Telnet server
[CHECK] System package telnetd is expected not to be installed

Profile Summary: 28 successful controls, 30 control failures, 1 control skipped
Test Summary: 122 successful, 60 failures, 2 skipped
```

Hardening mit Dev-Sec

Benötigte Ansible-Inhalte herunterladen:

```
$ ansible-galaxy collection install devsec.hardening
```

Beispielhaftes Ansible-Playbook:

```
- name: Harden servers
  hosts: node1
  become: true
  roles:
    - name: devsec.hardening.os_hardening
      sysctl_overwrite:
        # Enable IPv4 forwarding (needed for containers)
        net.ipv4.ip_forward: 1
```


Hardening mit Dev-Sec

Ausführen des Playbooks:

```
$ ansible-playbook hardening.yml
PLAY [Harden servers] *****
...
TASK [devsec.hardening.os_hardening : Configure selinux | selinux-01
ok: [node1]
PLAY RECAP *****
node1                : ok=53    changed=1    unreachable=0    f
```

Erneutes Auditieren:

```
$ inspec exec linux-baseline -t ssh://user@host
...
Profile Summary: 58 successful controls, 0 control failures, 1 contr
Test Summary: 182 successful, 0 failures, 2 skipped
```

testinfra

- `pytest`-Plugin für Infrastruktur-Tests
- Nützlich um den **tatsächlichen** Zustand von Servern zu überprüfen, die mittels Ansible oder SaltStack konfiguriert wurden
- ähnlich `serverspec`, aber in Python entwickelt
- unterstützt zahlreiche Verbindungsoptionen
 - local, paramiko, `ssh`, WinRM
 - Docker, Podman, LXC/LXD
 - Ansible, SaltStack
 - `kubectl`, OpenShift



testinfra

- kann auch ohne `pytest` benutzt werden
 - Integration in eigene Python-Anwendungen
 - optionale Nagios-kompatible Ausgabe
- **26** Module mit **88** Eigenschaften für verschiedene System-Komponenten
 - `environment`, `interface`, `mount_point`, `package`, ...
- einfacher Syntax, überschaubare Lernkurve
- gute Beispiele in der [Dokumentation](#)



testinfra

```
def test_firewall(host):
    """
    Check whether firewall is enabled
    """
    _service = host.service("firewalld.service")
    assert _service.is_enabled
    assert _service.is_running

def test_lab_users(host):
    """
    Check whether lab user exists and has valid home directory
    """
    user = host.user("sgiertz")
    home_dir = host.file(user.home)
    assert user.exists
    assert home_dir.is_directory
    assert home_dir.user == user.name

def test_httpd_listening(host):
    """
    Check whether httpd is listening
    """
    http_port = host.socket("tcp://0.0.0.0:80")
    assert http_port.is_listening
```

testinfra

- Tests können auch **parametrisiert** werden
- Größere Test-Suites in einzelne Dateien aufteilen
 - `tests_generic.py`
 - `tests_app1.py`
 - `tests_smoke.py`
- `pytest-xdist` für die Verteilung auf mehrere CPUs verwenden
- `sudo` wird nur mit der Option `NOPASSWD` unterstützt
 - kein eleganter Weg, um Passwort abzufragen
 - Ansible Inventory und `--force-ansible` sollen hier helfen, funktioniert für mich aber nicht

testinfra

```
@pytest.mark.parametrize("name,version", [
    ("lolcat", "1.5"),
    ("python3", "3.11"),
])
def test_packages(host, name, version):
    """
    Check whether requirements are installed
    """
    pkg = host.package(name)
    assert pkg.is_installed
    assert pkg.version.startswith(version)
```

pytest-xdist beschleunigt Tests

```
=====
platform linux -- Python 3.11.8, pytest-7.4.4, pluggy-1.0.0
rootdir:
plugins: testinfra-10.0.0
collected 36 items

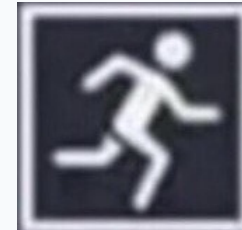
../../../../ansible/test_deployment.py .....

===== 36 passed in 283.69s (0:04:43) =====
```

```
=====
platform linux -- Python 3.11.8, pytest-7.4.4, pluggy-1.0.0
rootdir:
plugins: testinfra-10.0.0, xdist-3.5.0
8 workers [36 items]

.....

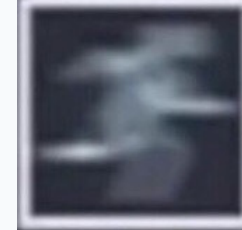
===== 36 passed in 45.19s =====
```



Walking



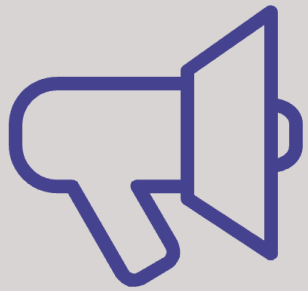
Running



Sprinting



python-xdist



Lessons Learned

Testen in der Produktion

- sinnvoll, wenn man es richtig macht
- Tests sollten nie Änderungen hervorrufen
- Hohe Systemlast vermeiden
 - Effizienter Code
 - Impact aktiver User
 - Generierung von Logs
- Health-Check APIs benutzter Anwendungen nutzen
- Tests in einer Testumgebung validieren, bevor diese in der Produktion ausgeführt werden!



Richtig testen

- Es reicht nicht aus, den Inhalt einer Konfigurationsdatei zu überprüfen
- Viel wichtiger ist es, den **tatsächlichen Zustand** einer Anwendung zu prüfen
 - lauscht der Server auf Port XYZ?
 - Antworten auf Anfragen semantisch überprüfen

Me: *checking TCP port 80*

Everybody else seeing the "Hello World" site:



Welcher Test ist besser?

Dieser?

```
def test_my_app(host):  
    web = host.service("httpd")  
    assert web.is_enabled  
    assert web.is_running
```

...oder dieser?

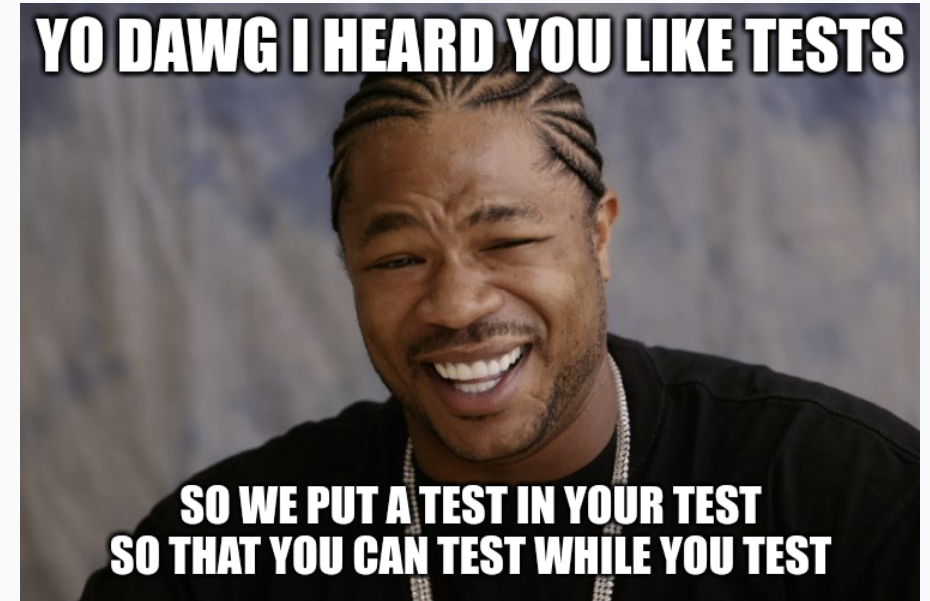
```
def test_socket(host):  
    web_sock = host.socket("tcp://0.0.0.0:80")  
    assert web_sock.is_listening  
  
def test_result(host):  
    _site = host.run(f"curl -L {site['url']}")  
    assert "myapp v1.0" in _site.output  
    assert _site.rc == 0
```

Prioritäten sind wichtig

- Zu viele Anforderungen erschweren die Einschätzung von Prioritäten
- KISS-Prinzip, mit Low Hanging Fruits anfangen
- funktionalen MVP anstreben, stetige iterative Verbesserungen
- [Paretoprinzip](#) hilft ebenfalls
 - auch *80-zu-20-Regel* genannt
 - 80% der Ergebnisse mit 20% des Aufwands
 - es gibt keine 100%, darf's etwas weniger sein?

Auch Tests testen

- automatische Tests sollten regelmäßig überprüft und aktualisiert werden
 - Systeme und Anforderungen ändern sich im Laufe der Zeit
- Das Simulieren von Test-Szenarien kann hier helfen
 - z.B. [Ansible Molecule](#)
 - Dokumentation regelmäßig überprüfen
- Das Verwenden von Bots wie [Renovate](#) kann die Wartung von Projekten vereinfachen



Exkurs: Molecule

- unterstützt bei der Entwicklung von Ansible-Inhalten
 - Collections, Rollen und Playbooks
- Automatisiert den vollständigen Testing-Workflow
 - VMs/Container erstellen und starten
 - Statische Code-Analyse (**linting**)
 - Inhalte ausführen (**converge**)
 - Tests mit **Verifier** ausführen (z.B. Ansible `assert` oder `testinfra`)
 - VMs/Container löschen
- Python-Paket, [molecule-plugins](#) erweitert die Funktionalität
 - Support für Container, Vagrant, OpenStack, etc.

Exkurs: Molecule

```
---
dependency:
  name: galaxy
driver:
  name: vagrant
lint: |
  yamllint .
  ansible-lint
  flake8
platforms:
  - name: opensuse-leap155
    box: opensuse/Leap-15.5.x86_64
    cpus: 4
    memory: 16384
provisioner:
  name: ansible
verifier:
  name: testinfra
```

Beispiel, um Ansible-Code via `testinfra` in einer openSUSE Leap 15.5-VM zu testen

Exkurs: Molecule

```
$ molecule list --format yaml
INFO          Running default > list
---

- Converged: 'false'
  Created: 'false'
  Driver Name: vagrant
  Instance Name: instance
  Provisioner Name: ansible
  Scenario Name: default
```

Anlegen einer Testumgebung, Ausführen des Codes und Testen des Zustands:

```
$ molecule create
$ molecule converge
$ molecule verify
```


Dokumentation ist wichtig

- Gute Dokumentationen erklären die Motivation
 - Warum wurde dieser Test erstellt?
 - Welche Parameter werden überprüft?
 - In welchen Zuständen können die einzelnen Zustände sein?
- Dokumentation kann direkt aus dem Quellcode heraus **generiert werden**
 - `doxygen` , `sphinx`
- Gängige Test-Frameworks unterstützen Protokolle wie `junit-xml`
 - bessere Sichtbarkeit von Tests und Veränderungen



Dokumentation ist wichtig

Auszug einer GitLab-Pipeline:

```
run tests:
  stage: test
  script:
    - cd test
    - pytest --hosts="ansible://instances" \
      --ansible-inventory=inventory.ini \
      --junit-xml=pytest-validator.xml
  artefacts:
    when: always
    paths:
      - test/pytest.xml
    reports:
      junit: test/pytest.xml
  allow_failure: true
```

Dokumentation ist wichtig

SVA

CODE HUB

2

12

11

Search or go to...

Project

- V Vaultinator
- Pinned
- Issues0
- Merge requests0

Manage>

Plan>

Code>

Build>

Pipelines

Jobs

Pipeline schedules

Test cases

Artifacts

Deploy>

Operate>

Monitor>

Analyze>

Schmitt, Sebastian / Vaultinator / Pipelines / #310456

WIP

Blocked Schmitt, Sebastian created pipeline for commit fdc6b715

For main

latest GO 6 jobs

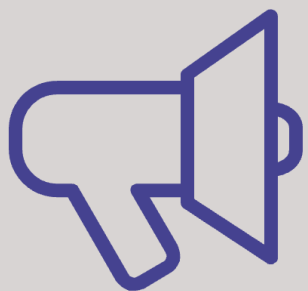
PipelineJobs6Tests29

< run tests

29 tests0 failures0 errors100% success rate11.69s

Tests

Suite	Name	Filename	Status	Duration	Details
test_extended_local	test_apparmor_selinux[ansible://vault-0]		✓	1.00ms	View details
test_extended_local	test_ulimits[ansible://vault-0]		✓	1.00ms	View details
test_extended_local	test_containers[ansible://vault-0]		✓	1.00ms	View details
test_baseline_remote	test_audit_logging[ansible://vault-0]		✓	0.00ms	View details
test_baseline_remote	test_vault_version[ansible://vault-0]		✓	0.00ms	View details



Und nun?

Links

Bei welchem Projekt könntest Du Tests einsetzen?

Nachlese

- [DevSec Hardening Framework](#)
- [ansible-lockdown](#)
- [RHEL 9 Security Hardening Guide](#)
- [SLES 15 Security and Hardening Guide](#)
- [goss](#)
- [InSpec](#)
- [testinfra](#)



Danke für die Aufmerksamkeit

