Getting Started with Vane

What is Vane?

Vane is an open source, network validation tool which runs tests against Arista networking devices, this is achieved by connecting to devices on a given network, executing commands and performing tests against their output. Vane eliminates repetitive testing tasks by automating network validation which can take users months to complete.

Vane places great emphasis on its ease of use, its intention is to minimize the need for system operators to edit source code by the use of dynamically passing information to its test cases. This is achieved by using yaml files which contain the information to be passed as a parameter to a given test case.

Another important feature of the tool is its ability to report test case output in various formats, including JSON, HTML, word document or excel spreadsheets. These different outputting formats allow for improved readability of test case output and also structured data which is accessible to analyze test output.

Technologies in Vane

Vane at its core is a Python project, python classes are responsible for the parsing of its command line arguments, setting up tests and the execution of tests, and also for the reporting of test output. Since Vane is based on Python, developers experienced with Python can easily write or extend test cases.

PyTest is utilized for the reporting aspect of the product, it supports for easily running test cases with its existing functionality, all Vane test cases respect PyTest syntax. Vane can be thought of as a wrapper around PyTest with enhanced functionality. Arista development is focused on improving the user experience and making it easier to write test cases.

Installing Vane

Vane can be installed in two different ways, via a docker container or a virtual environment.

Docker Installation

Step 1: Clone the repository
```
> git clone git@github.com:aristanetworks/vane.git
```
Step 2 : Create docker container and install vane requirements
```
> make dev
```
Step 3: Install vane
```
> make install
```

Virtual Environment Installation

Step 1: Clone the repository
```
> git clone git@github.com:aristanetworks/vane.git
```

Step 2: Create & activate virtual environment
```
> python3 -m venv venv
> source venv/bin/activate
```

Step 3: Install Vane
```
> cd vane
> pip install -r requirements.txt
> make install
```

Vane Directory Structure

As Vane is a Python application, it uses best practices to reflect a Python application layout.

The **reports** directory includes the different reports generated by Vane in .docx and html format along with some .yaml results
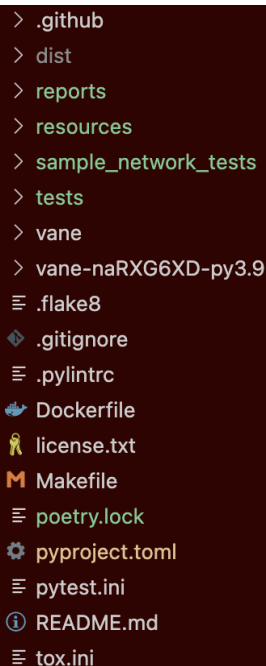
The **sample_network_tests** directory has test cases which Vane runs against the devices in the given network topology. It also includes a sample definitions.yaml and sample duts.yaml for initial vane execution

The **tests** directory has the unit tests which run against Vane to ensure correct functionality of the application itself.

The **vane** directory includes the python files which are used to execute Vane, such as parsing of command line arguments, utilities to run the test cases and report relevant results.

The **vane-virtual environment** directory is the virtual environment that poetry creates when you run the *poetry install command*. It initializes a virtual environment with all the dependencies and requirements mentioned in the pyproject.toml.

The **top level directory** includes the following items in addition to those listed above:

> .github
> dist
> reports
> resources
> sample_network_tests
> tests
> vane
> vane-naRXG6XD-py3.9
≡ .flake8
◈ .gitignore
≡ .pylintrc
🐳 Dockerfile
📜 license.txt
M Makefile
≡ poetry.lock
⚙ pyproject.toml
≡ pytest.ini
ⓘ README.md
≡ tox.ini

.github folder which contains development pipeline files

dist folder has the wheel and sdist distribution of vane

resources folder containing assets (svg's) used by vane

Project setup files such as poetry.lock, pytest.ini, tox.ini, pyproject.toml and .gitignore

Linting files for formatting purposes such as .flake8 and .pylintrc

Documentation files such as README.md, license.txt, MakeFile, Dockerfile

Vane Tests Structure

Test files in Vane are python files which are run using PyTest. The functions within the files respect PyTest syntax allowing the interpreter to detect and run given test cases.

Each Vane test directory always has an accompanying **test definition** file, this is an yaml file which serves information to a test case. Data within this file should include the expected output of the test, any commands or configurations which need to be run prior to the test, a filter to denote which devices are specific to the test, and a brief description of the test case. This file allows for dynamically passing information to a test case rather than hard coding information into the python file, it has the benefit of being user friendly in a structured data format allowing for less-knowledgeable operators to modify their test cases.

Directory Structure:

```
tests/systests/system
├── __init__.py
├── test_definition.yaml
└── test_system.py
```

test_definition.yaml:

```yaml
name: test_system.py
testcases:
  - name: test_if_there_is_agents_have_crashed_on_
    description: Verifies the agents logs crash is empty
    show_cmd: null
    # Number of exceptabl crash logs
    expected_output: 0
  - name: test_if_eos_version_is_correct_on_
    description: Verifies EOS version running on the device
    show_cmd: show version
    # EOS version name
    expected_output: 4.24.2F
```

test_system.py - EOS Version Test:

```python
def test_if_eos_version_is_correct_on_(self, dut, tests_definitions):
    """Verifies EOS version running on the device

    Args:
        dut (dict): Encapsulates dut details including name, connection
        tests_definitions (dict): Test parameters
    """

    tops = tests_tools.TestOps(tests_definitions, TEST_SUITE, dut)

    tops.actual_output = dut["output"][tops.show_cmd]["json"]["version"]
    tops.test_result = tops.actual_output == tops.expected_output

    tops.output_msg = (
        f"On router |{tops.dut_name}| EOS version is "
        f"|{tops.actual_output}%|, version should be "
        f"|{tops.expected_output}%|"
    )
    tops.comment = (
        "TEST EOS version running on the device on "
        f"|{tops.dut_name}|.\n"
        f"GIVEN version is |{tops.expected_output}|.\n"
        f"WHEN version is |{tops.actual_output}|.\n"
        f"THEN test case result is |{tops.test_result}|.\n"
        f"OUTPUT of |{tops.show_cmd}| is:\n\n{tops.show_cmd_txt}"
    )

    tops.post_testcase()

    assert tops.actual_output == tops.expected_output
```

Vane Master Definition file

Vane master definition file has been added to avoid editing the test definition yaml files in each directory. If this file has been added to a test case repository, then it provides the user a single place to edit all the variables defined for all the tests in the test definition yaml files of a repository. The master definition file is again a yaml file containing two sections:

- Common data
- Test case data

Common data is the data which can be referenced by several test cases, for ex: server login information, ntp server information etc.

Test case data is the data specific to a test case.

The data from the master definition file is used to generate test definition yaml files in the pytest directories. Each of the pytest directories would have a jinja2 test definition file where all the test variables would be replaced by jinja2 variables. This jinja2 test definition file is then rendered using the master definition file, which creates the test definition yaml file for each pytest directory.

Generating test definitions using master definition file

The user can control generation of test definition yaml files using the vane 'definitions.yaml' file:

```yaml
parameters:
  eapi_file: eapi.conf
  eapi_template: eapi.conf.j2
  eos_conn: eapi
  excel_report: null
  html_report: reports/report
  json_report: reports/report
  mark:
  processes: null
  report_dir: reports
  results_file: result.yml
  results_dir: reports/results
  setup_show: false
  show_log: show_output.log
  stdout: false
  test_cases: All
  test_dirs:
  - tests/nrfu_tests/
  report_test_steps: true
  generate_test_definitions: false
  master_definitions: master_def.yaml
  template_definitions: tests_definitions.yaml.j2
  test_definitions: test_definition.yaml
  verbose: true
```

The turn on generation of test definition yaml files using master definition file, make the 'generate_test_definitions' variable shown above to 'true'. The 'master_definitions' variable highlighted above specifies which master definition file to use. The 'template_definitions' specify which test definition jinja2 template file to look for in each pytest directory. And lastly, the 'test_definitions' variable specifies the name of the test definition yaml file which is to be generated. This would be the same file that would be used by vane to input data to the test cases.

Executing Vane

The application is run through the command line by passing in different arguments as needed. Using the command '**vane --help**' shows the different available arguments that can be used.

Below are the details of some of the necessary flags (arguments), what they stand for and how they can be included in order to run vane.

1. **--definitions_file definitions.yaml**

The Vane definitions file serves as an informative file to Vane, it includes different information such as

- **test_dirs**: the source directory which contains the test cases to be run,
- **report_dir**: the directory where report files generated by Vane should be stored,
- **test_cases**: list of test cases within the test_dir which need to be run,
- **markers:** markers help group various test cases executed by pytest
- and the location of relevant **configuration files**

Variables in this file can be edited by the operator to cater to the application execution.

If while running vane you do not include this argument as follows:

vane –definitions_file definitions.yaml (where definitions.yaml is the relative path to your definitions file, the one in vane repo is at: vane/sample_network_tests/definitions.yaml) then vane defaults to using the definitions file mentioned in the vane/config.py file

Example of definitions.yaml:

```
parameters:
  Explorer (⇧⌘E) : tests/unittests/fixtures/eapi.conf
  eapi_template: tests/fixtures/templates/eapi.conf.j2
  eos_conn: eapi
  excel_report: null
  html_report: tests/systests/fixtures/reports/report
  json_report: tests/systests/fixtures/reports/report
  mark: demo
  processes: null
  report_dir: tests/systests/fixtures/reports
  results_file: result.yml
  results_dir: tests/systests/fixtures/reports/results
  setup_show: false
  show_log: show_output.log
  stdout: false
  test_cases: All
  test_dirs:
  - tests/systests
  test_definitions: tests_definitions.yaml
  verbose: true
  spreadsheet: tests/fixtures/spreadsheets/PS-LLD-Questionnaire-Template.xlsx
  xcel_definitions: tests/fixtures/spreadsheets/xcel_definitions.yaml
  xcel_schema: tests/fixtures/spreadsheets/xcel_schema.yaml
```

2. –duts_file duts.yml

The Vane duts file includes a list of all devices that Vane should run its test cases against, it includes relevant information for each DUT (device under test) such as their hostname and their access credentials. Operators should edit this file to include the devices that they would like to run Vane against.

```yaml
duts:
- mgmt_ip: 10.255.74.38
  name: BL1
  neighbors:
  - neighborDevice: leaf1
    neighborPort: Ethernet1
    port: Ethernet1
  - neighborDevice: leaf2
    neighborPort: Ethernet1
    port: Ethernet2
  password: cvp123!
  transport: https
  username: cvpadmin
  role: leaf
- mgmt_ip: 10.255.22.26
  name: BL2
  neighbors:
  - neighborDevice: leaf1
    neighborPort: Ethernet1
    port: Ethernet1
  - neighborDevice: leaf2
    neighborPort: Ethernet1
    port: Ethernet2
  password: cvp123!
  transport: https
  username: cvpadmin
  role: leaf
```

This file can be generated by using the following command:

vane –generate-duts-file topology.yaml inventory.yaml – (1)

topology.yaml : this file represents the topology of the virtual lab you will run Vane against. For demoing purposes we have a virtual lab deployed in ACT within the EOS+ tenant called: Vane Demo Lab. You can get the topology file from under there.

inventory.yaml : this file can be downloaded from ACT from the view which reflects your deployed lab.

Once you have these two files in your vane directory, you can use the above command (1) to generate a duts file which Vane will run against.

Note: For initial demo, we have a sample duts.yaml file in the sample_network_tests folder configured on the Vane Demo Lab

If while running vane you do not include this argument as follows:

vane –duts-file duts.yaml (where duts.yaml is the relative path to your duts file, the one you created using the –generate-duts-file command above or you can use the sample on in sample_network_tests folder) then vane defaults to using the duts file mentioned in the vane/config.py file

There are a few other flags and descriptions on how to use them which you can explore further, but the ones described above are the necessary ones to get started.

Executing Vane

Command to Run Vane: vane

NOTE: Ensure the DEFINITIONS_FILE and DUTS_FILE variables in config.py are pointing to the correct location for the respective files if you want to run vane using "**vane**" command, if not you might have to explicitly mention the arguments and give the locations in the command as follows:

vane –definitions-file sample_network_tests/definitions.yaml –duts-file sample_network_tests/ duts.yaml

(NOTE: Ensure the ACT lab is running, when you run Vane)

Below is a sample run of vane:

Note: If the test cases get executed correctly, it implies vane has been set up correctly, the failure of test cases in itself does not imply an error on vane's execution side of things.

Viewing Reports

After Vane has executed successfully, test case reports get generated and populated in the reports folder which exists in the outermost directory of vane. You can view these reports in multiple formats, including json, .docx, html. A sample of one such .docx report is listed below. These reports offer detailed information on the test cases such as test case procedure, input, expected output, pass/fail result, and other relevant observations.

3. Detailed Test Suite Results: Mgmt

3.1 Test Case: Test ntp functional

3.1. 1 DUT: DLFW3
TEST IDENTIFIER: TN4.12

TEST CASE NAME: test_ntp_functionality

DESCRIPTION: Verification of NTP functionality

ASSUMPTIONS:

- No assumptions have been made

DEVICE UNDER TEST: DLFW3

EXTERNAL SYSTEMS:

- No external systems are being used

CONFIGURATION:

```
Network configuration is unchanged
```

PROCEDURE:

1. Run show command 'show ntp status' on dut
2. Verify NTP status is 'synchronized'
3. Verify NTP server has correct IP address

INPUT:

```
ntp_server_ip: 64.27.23.17
ntp_status: synchronise
```

DUT OUTPUT:

```
DLFW3# show ntp status:

synchronised to NTP server (64.27.23.177) at stratum 3
   time correct to within 45 ms
   polling server every 64 s
```

EXPECTED OUTPUT: NTP server is reachable and the NTP clock is in synchronous mode.

OBSERVATION: NTP server status is NOT synchronise. NTP server status is synchronised. NTP server IP address is incorrectly set to 64.27.23.177. NTP server IP address should be set to 64.27.23.17.

PASS/FAIL: FAIL

COMMENT: