fwunit Documentation

Release unknown

Dustin J. Mitchell

Jun 07, 2017

Contents

Any developer worth their salt tests their software. The benefits are many:

- Exercise the code
- Reduce ambiguity by stating the desired behaviors twice (in the implementation, in the tests, and maybe even a third time in the documentation)
- Enable code refactoring without changing expected behavior

With `fwunit`, you can do the same for security policies on your network.

CHAPTER 1

Contents

Principle of Operation

Testing policies is a two-part operation: first fetch and process the data from all of the applicable devices, constructing a set of *rules*. Then run the tests against the rules.

This package handles the first part entirely, and provides a set of utility functions you can use in your tests. You can write tests using whatever Python testing framework you like. We recommend nose.

Installation

To install, set up a Python virtualenv and then run

```
pip install fwunit[srx,aws]
```

where the bit in brackets lists the systems you'd like to process (see *Supported Policy Types*).

Processing Policies

To gather data about your network flows, you will need to define one or more "sources" in a configuration file. Each source describes a set of flow configurations for fwunit to convert into its internal representation and store. For example, you might define one source for each distinct firewall in your organization, or for each distinct AWS account.

You'll then run `fwunit` in the directory containing the configuration file, and it will process the policies from each source and write them to disk, ready for analysis.

You can pass one or more source names to `fwunit` to only process those sources. Otherwise it processes all sources, ordered by their dependencies.

Configuration File

The `fwunit` command processes a YAML-formatted configuration file describing a set of "sources" of rule data. Each top-level key describes a source, and must have a `type` field giving the type of data to be read – see "Supported Systems", above.

```
aws_releng:
    type: aws
    output: aws_releng.json
    dynamic_subnets: [build, test, try, build.servo, bb]
    regions: [us-east-1, us-west-1, us-west-2]
```

Each must also have an `output` field giving the filename to write the generated rules to (relative to the configuration file).

The source may optionally have a `require` field giving a list of other sources which should be processed first.

Any additional fields are passed to the policy-type plugin. See the documentation of those plugins for more information.

Application Maps

Each policy type comes with its own way of naming applications: strings, protocol/port numbers, etc.

An "application map" is used to map these type-specific names to common names. This is invaluable if you are combining policies from multiple types, e.g., AWS and SRX.

To set this up, add an `application-map` key to the source configuration, with a mapping from type name to common name. For example:

Note that you *cannot* combine multiple applications into one using an application map, as this might result in overlapping rules.

Supported Policy Types

fwunit can read policies from several sources:

Amazon EC2 Security Groups

Setup

Install fwunit with the `aws` tag:

```
pip install fwunit[aws]
```

Set up your `~/.boto` with an account that has access to EC2 and VPC administrative information.

In your source configuration, include `dynamic_subnets` listing the names or id's of all dynamic subnets (see below). Also include a `regions` field listing the regions in which you have hosts.

You can include the credentials for an IAM user in the configuration. If this is omitted, boto's normal credential search process will apply, including searching `~/.boto` and instance role credentials.

```
my_aws_stuff:
    type: aws
    output: my_aws_stuff.pkl
    dynamic_subnets: [workers]
    regions: [us-east-1, us-west-1]
    credentials:
        access_key: "ACCESS KEY"
        secret_key: "SECRET KEY"
```

Security Policy

The user accessing Amazon should have the following security policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

Assumptions

This processing makes some assumptions about your EC2 layout. These worked for us in Mozilla Releng, but may not work for you.

- Network ACLs are not in use
- All traffic is contained in subnets in one or more VPCs.
- Each subnet is either *per-host* or *dynamic*, as described below.
- All traffic from unoccupied IPs in per-host subnets is implicitly permitted.
- Subnets with the same name are configured identically. Such subnets are often configured to achieve AZ/region separation.

The Release Engineering AWS environment contains two types of instances, which always appear in different subnets. Long-lived instances sit at a single IP for a long time, acting like traditional servers. The subnets holding such instances are considered "per-host" subnets, and the destination IPs for `fwunit` rules are determined by examining the IP addresses and security groups of the instances in the subnets. All traffic to IPs not assigned to an instance is implicitly denied.

The instances that perform build, test, and release tasks are transient, created and destroyed as economics and load warrant. Subnets containing such instances are considered "dynamic", and a security group that applies to any instance in the subnet is assumed to apply to the subnet's entire CIDR block. This means that these subnets must contain at least one active host.

Juniper SRX

This source type uses SSH with a username and password to connect to a Juniper SRX firewall. It only runs 'show' commands, so read-only access is adequate.

Setup

Install fwunit with the `srx` tag:

```
pip install fwunit[srx]
```

Add a source to your `fwunit.yaml` looking like this:

```
myfirewall:
    type: srx
    output: myfirewall.pkl
    firewall: fw1.releng.scl3.mozilla.com
    ssh_username: fwunit
    ssh_password: sekr!t
```

The `firewall` config gives a hostname (or IP) of the firewall that accepts SSH connections. `ssh_username` and `ssh_password` are the credentials for the account.

The process of downloading and processing policies can be very slow, depending on the complexity of your policies.

Assumptions

This processing makes the following assumptions about your network

- Rule IPs are limited by the to- and from-zones of the original policy, so given a "from any" policy with from-zone ABC, the resulting rule's `src` will be ABC's IP space, not 0.0.0.0/0. Zone spaces are determined from the route table, and thus assume symmetrical forwarding.
- All directly-connected networks are considered to permit all traffic within those networks, on the assumption that the network is an open L2 subnet.
- Policies allowing application "any" are expanded to include every application mentioned in any policy.

Policy Combiner

A large organization will have multiple policy sources, perhaps in different regions or of different types. Before you can write tests and reason about the overall flows, these must be combined into a single rule set.

A complicating factor is that, depending on how flows are routed between particular address spaces, they may flow through an arbitrary set of firewalls. For example, traffic from San Francisco to New York may flow through the Colorado and Chicago firewalls, while traffic from Iowa city only flows through the Chicago firewall.

To accomplish this, the policy combiner requires you to separate IP addresses into "address spaces", then define the sources defining the rules between these spaces.

For example:

```
enterprise:
    type: combine
    output: enterprise.json
    require: [fw.dca, fw.lax, fw.ord]
```

```
    address_spaces:
        dca: [10.10.0.0/16, 10.15.0.0/14]
        ord: 192.168.0.0/24
        lax: 172.16.2.0/24
    routes:
        # all traffic to or from dca passes through its firewall
        'dca <-> *': fw.dca
        # similarly for the other sites
        'ord <-> *': fw.ord
        'lax <-> *': fw.lax
        # traffic from dca to lax passes through ord too, but not the
        # reverse
        'dca -> lax': fw.ord
        # and all external traffic is via lax (and ord for dca)
        'dca <-> unmanaged': [fw1.ord, fw1.lax]
        'ord <-> unmanaged': fw1.lax
```

If (as in this example) the address spaces do not cover the entirety of IPv4, then an address space named `unmanaged` is automatically created to cover the remainder.

The `routes` mapping defines the set of rule sources applied between pairs of IP spaces. The `*` wildcard matches all address spaces (including `unmanaged`). The `<->` symbol is equivalent to listing two routes, one in each direction. Where multiple routes match, all named rule sources are applied.

Querying

Aside from writing unit tests, you can query against a rule source with `fwunit-query`.

For example:

```
fwunit-query permitted enterprise 10.10.1.1 192.168.1.1 ssh
Flow permitted
```

See the script's `--help` for more detail.

Diffing

Two compare two rulesets, use `fwunit-diff`. For example:

```
$ fwunit-diff yesterday.json my-network
+ ssh IPSet([IP('172.16.3.0/24')]) -> IPSet([IP('10.90.110.0/23')])
```

The two sources for comparison can be the names of sources defined in `fwunit.yaml`, or filenames (e.g., to backup copies).

Writing Tests

Flow tests are just like software unit tests: they make assertions about the state of the system under test. In the case of flow tests, that means asserting that traffic can, or cannot, flow between particular systems.

Here's how to write flow tests.

Example

The following might be in a file named *test_puppet.py*.

```
from fwunit import TestContext
from fwunit import IP, IPSet

tc = TestContext('my-network')

# hosts

internal_network = IPSet([IP('192.168.1.0/24'), IP('192.168.13.0/24')])
external_network = IPSet([IP('0.0.0.0/0')]) - internal_network
puppetmasters = IPSet([IP(ip) for ip in
    '192.168.13.45',
    '192.168.13.50',
])

# tests

"""
Puppetmasters serve puppet catalogs and data to clients over 'puppet',
'http', and 'https'.  All hosts in the internal network should have access.
"""

def test_puppetmaster_access():
    """The entire internal_network can access the puppet masters."""
    for app in 'puppet', 'http', 'https':
        tc.assertPermits(internal_network, puppetmasters, app)

def test_puppetmaster_no_other_apps():
    """Access to puppetmasters is limited to puppet, http, and https"""
    tc.assertAllApps(IPSet([IP('0.0.0.0/0')]), puppetmasters,
                     ['puppet', 'http', 'https'])

def test_puppetmaster_limited():
    """The exteernal networks cannot access the puppet masters."""
    for app in 'puppet', 'http', 'https':
        tc.assertDenies(external_network, puppetmasters, app)
```

Running this test is as simple as

```
$ nosetests test_puppet.py
```

Loading Rules

Before you can test anything, you'll need to load the rules created with the `fwunit` command into memory. It's safe to do this individually in each test script, as the results are cached.

```
from fwunit import TestContext

tc = TestContext('source-name')
```

The `TestContext` class uses `fwunit.yaml` in the current directory to look up the proper source file for the given source name.

IPs and IPSets

The `IP` and `IPSet` classes come from IPy, with minor changes.

The `IP` class represents a single IP or CIDR range:

```
from fwunit import IP
server = IP('10.11.12.33')
subnet = IP('10.11.12.0/23')
```

When you need to reason about a non-contiguous set of addresses, you need an `IPSet`. This is really just a list of `IP` instances, but it will remove duplicates, collapse adjacent IPs, and so on.

```
from fwunit import IP, IPSet
db_subnets = IPSet([IP('10.11.12.0/23'), IP('10.12.12.0/23')])
```

In general, tests expeect `IPSets`, but you can pass `IP` instances or even bare strings and they will be converted appropriately.

Tests

Once you have the rules loaded, you can start writing test methods:

```
internal_network = IPSet([IP('192.168.1.0/24'), IP('192.168.13.0/24')])

puppetmasters = IPSet([IP(ip) for ip in
    '192.168.13.45',
    '192.168.13.50',
])

def test_puppetmaster_access():
    for app in 'puppet', 'http', 'https':
        tc.assertPermits(internal_network, puppetmasters, app)
```

Utility Methods

The `TestContext` class provides a number of useful functions for testing. Each method logs verbosely, so test failures should have plenty of data for debugging.

class `funit.analysis.testcontext.`**`TestContext`**(*source_name*)

Parameters **`source_name`** – fwunit source from which to load rules

`assertDenies`(*src*, *dst*, *app*)

Parameters

- **`src`** – source IPs
- **`dst`** – destination IPs
- **`apps`** (*`list or string`*) – application names

Assert that traffic is denied from any given source IP to any given destination IP for all given applications.

`assertPermits`(*src*, *dst*, *apps*)

Parameters

- **`src`** – source IPs

- **dst** – destination IPs
- **apps** (*list or string*) – application names

Assert that all given applications are allowed from any given source IP to any given destination IP.

Note that `assertDenies` and `assertPermits` are not quite opposites: if application traffic is allowed between some IP pairs, but denied between others, then both methods will raise `AssertionError`.

sourcesFor(*dst*, *app*, *ignore_sources=None*)

Parameters

- **dst** – destination IPs
- **app** – application
- **ignore_sources** – source IPs to ignore

Return an IPSet with all sources for traffic to any IP in dst on application app, ignoring flows from ignore_sources.

This is useful for assertions of the form "access to X is only allowed from Y and Z".

allApps(*src*, *dst*, *debug=False*)

Parameters

- **src** – source IPs
- **dst** – destination IPs
- **debug** – if True, log the full list of matching flows

Return a set of applications with access form src to dst.

This is useful for verifying that access between two sets of hosts is limited to a short list of applications.

Note that if *any* application is allowed from `src` to `dst`, this method will return `set(['any'])` rather than enumerating the (infinite) set of allowed applications.

assertAllApps(*src*, *dst*, *apps*, *debug=False*)
:param src source IPs :param dst: destination IPs :param apps: expected list of applications :param debug: if True, log the full list of matching flows

Verify that the set of applications with access from any host in `src` to any host in `dst` is `apps`.

This is useful for verifying that other tests have covered all of the open applications. The same warning as for *`allApps()`* applies here for rules allowing any application.

Implementation Notes

IP Objects

This tool uses IPy to handle IP addresses, ranges, and sets. However, it extends that functionality to include some additional methods for `IPSets` as well as an `IPPairs` class to efficiently represent sets of IP pairs.

All of these classes can be imported directly from `fwunit`.

Rules

The output of the processing step is a JSON-formatted object. The `rules` key gives a list of rule objects, each of which has keys

- `src` - a list of source IP ranges
- `dst` - a list of destination IP ranges
- `app` - the name of the permitted application
- `name` - the name of the rule

The rules are normalized as follows (and this is what consumes most of the time in processing):

- For a given source and destination IP and application, exactly 0 or 1 rules match; stated differently, there's no need to consider rules in order.
- If traffic matches a rule, it is permitted. If no rule matches, it is denied.
- Policies allowing any application are represented by explicit rules for each known application, with the addition of rules with application '@@other' to represent the unknown applications.

Loading Source Objects

Rule sets are embedded in *Source* objects, which provide a set of useful methods for analysis. In a testing environment, rule sets are loaded via `TestContext`; this section describes access to rules within fwunit itself.

To get a source object, you will first need a config, which can be retrieved from *load_config()*:

`fwunit.analysis.config.`**`load_config`**(*filename="fwunit.yaml"*)

Parameters **`filename`** – the configuration filename to load

Returns a config object

Load a configuration file. As a side-effect, this function chdir's to the configuration directory.

The function operates on the assumption that a single process will only ever reference one configuration, and thus caches the configuration after the first call. Subsequent calls with the same filename will return the same object. Subsequent calls with a different filename will raise an exception.

With the config object in hand, call *load_source()*:

`fwunit.analysis.sources.`**`load_source`**(*config*, *source*)

Parameters

- **`config`** – a config object
- **`source`** – the name of the source to load

Returns a source object

Return type *Source*

Load the ruleset for the given source. The `source` parameter can be a source name from the configuration, or a filename.

Rulesets are cached globally to the process.

Using Source Objects

class `fwunit.analysis.sources.`**`Source`**

The data from a particular source in `fwunit.yaml`, along with some analysis methods.

`rulesForApp(app)`:

Parameters **`app`** – application name

Returns list of rules

Get the rules for the given app, or if no such app is known, for `@@other`.

`rulesDeny`(*src*, *dst*, *apps*)

Parameters

- **`src`** – source IPs
- **`dst`** – destination IPs
- **`apps`** (*`list or string`*) – application names

Returns True if the rules deny all traffic from *src* to *dst* via all given *apps*; otherwise False.

`rulesPermit`(*src*, *dst*, *apps*)

Parameters

- **`src`** – source IPs
- **`dst`** – destination IPs
- **`apps`** (*`list or string`*) – application names

Returns True if the rules allow all traffic from *src* to *dst* via all given *apps*; otherwise False.

Note that `rulesdeny(..)` is not the same as `not rulesPermit(..)`: if some – but not all – traffic is permitted from *src* to *dst*, then both methods will return False.

`allApps`(*src*, *dst*, *debug=False*)

Parameters

- **`src`** – source IPs
- **`dst`** – destination IPs
- **`debug`** – if True, log the full list of matching flows

See `allApps()`.

`sourcesFor`(*dst*, *app*, *ignore_sources=None*)

Parameters

- **`dst`** – destination IPs
- **`app`** – application
- **`ignore_sources`** – source IPs to ignore

See `sourcesFor()`.

Index