Trigger Documentation

Release 1.0.0.100

**Jathan McCollum, Eileen Tschetter, Mark Ellzey Thomas, Michael

October 04, 2012

CONTENTS

"Go ahead, pull it!"

CHAPTER

ONE

ABOUT

Trigger is a Python framework and suite of tools for interfacing with network devices and managing network configuration and security policy. Trigger was designed to increase the speed and efficiency of network configuration management.

Trigger's core device interaction utilizes the freely available Twisted event-driven networking engine. The libraries can connect to network devices by any available method (e.g. telnet, SSH), communicate with them in their native interface (e.g. Juniper JunoScript, Cisco IOS), and return output. Trigger is able to manage any number of jobs in parallel and handle output or errors as they return.

1.1 Motivation

Trigger was created to facilitate rapid provisioning and automation of firewall policy change requests by Network Security. It has since expanded to cover all network device configuration.

The complexity of the network was increasing much more quickly than the amount of time we had to spend on administering it, both because AOL's products and services were becoming more sophisticated and because we were continually expanding infrastructure. This pressure created a workload gap that had be filled with tools that increased productivity.

Pre-Trigger tools worked only for some common cases and required extensive knowledge of the network, and careful attention during edits and loads. Sometimes this resulted in a system-impacting errors, and it routine work more dangerous and unrewarding than it should have been.

With the high number of network devices on the AOL network Trigger has become invaluable to the performance and reliability of the AOL network infrastructure.

1.2 History

Trigger was originally originally written by the AOL Network Security team and is now maintained by the Network Engineering organization.

Once upon a time Trigger was actually called **SIMIAN**, a really bad acronym that stood for **System Integrating Management of Individual Access to Networks**. It has since outgrown its original purpose and can be used for any network hardware management operations, so we decided to ditch the acronym and just go with a name that more accurately hints at what it does.

CHAPTER

TWO

SUPPORTED VENDORS

Trigger currently supports devices manufactured by the following vendors:

- Arista Networks
  - All 7000-family platforms
- Brocade Networks
  - MLX routers and VDX switches
- Cisco Systems
  - IOS-based platforms only including all Catalyst switches and GSR/OSR routers
- Dell
  - PowerConnect switches
- Foundry/Brocade
  - All router and switch platforms (NetIron, ServerIron, et al.)
- Juniper Networks
  - All router and switch platforms running Junos
  - NetScreen firewalls running ScreenOS (Junos not yet supported)
- Citrix Systems
  - NetScaler web accelerator switches (SSH only, no REST/SOAP yet)

CHAPTER

THREE

INSTALLATION

To install Trigger, please check out the installation docs!

3.1 Installation

This is a work in progress. Please bear with us as we expand and improve this documentation. If you have any feedback, please don't hesitate to contact us!!

3.1.1 Dependencies

In order for Trigger's core functionality to work, you will need the primary pieces of software:

- the Python programming language (version 2.6 or higher);
- the `setuptools` packaging/installation library;

- the Redis key-value server (and companion Python interface);
- the `IPy` IP address parsing library;
- the PyASN1 library;
- the PyCrypto cryptography library;
- and the Twisted event-driven networking engine.

Trigger has a tricky set of dependencies. If you want to take full advantage of all of Trigger's functionality, you'll need them all. If you only want to use certain parts, you might not need them all. Each dependency will list the components that utilize it to help you make an informed decision.

Please read on for important details on each dependency – there are a few gotchas.

Python

Obviously Trigger requires Python. Only version 2.6 is supported, but Python 2.7 should be just fine. There is currently no official support for Python 3.x. We cannot yet say with confidence that we have worked out all of the legacy kinks from when Trigger was first developed against Python 2.3.

setuptools

Setuptools comes with some Python installations by default; if yours doesn't, you'll need to grab it. In such situations it's typically packaged as `python-setuptools`, `py26-setuptools` or similar. Trigger will likely drop its setuptools dependency in the future, or include alternative support for the Distribute project, but for now setuptools is required for installation.

PyASN1

PyASN1 is a dependency of Twisted Conch which implements Abstract Syntax Notation One (ASN.1) and is used to encode/decode public & private OpenSSH keys.

PyCrypto

PyCrypto is a dependency of Twisted Conch which provides the low-level (C-based) encryption algorithms used to run SSH. There are a couple gotchas associated with installing PyCrypto: its compatibility with Python's package tools, and the fact that it is a C-based extension.

Twisted

Twisted is huge and has a few dependencies of its. We told you this was tricky! To make things easier, please make sure you install the full-blown Twisted source tarball. You especially need Twisted Conch, which is used to run SSH.

Used by:

- `trigger.cmds`
- `trigger.twister`

Redis

Trigger uses Redis as a datastore for ACL information including device associations and the integrated change queue. Please follow the instructions on the Redis site to get Redis running.

If you're using Ubuntu, it's as simple as:

```
sudo apt-get install redis-server
```

The python redis client is required to interact with Redis.

Trigger currently assumes that you're running Redis on localhost and on the default port (6379). If you would like to change this, update `REDIS_HOST` in `settings.py` to reflect the IP address or hostname of your Redis instance.

Used by:

- `trigger.acl.autoacl`
- `trigger.acl.db`
- `trigger.acl.tools`
- `trigger.netdevices`

IPy

IPy is a class and tools for handling of IPv4 and IPv6 addresses and networks. It is used by Trigger for parsing and handling IP addresses.

Used by:

- `trigger.acl.db`
- `trigger.acl.parser`
- `trigger.acl.tools`
- `trigger.cmds`
- `trigger.conf.settings`
- `trigger.netscreen`

pytz

pytz is an immensely powerful time zone library for Python that allows accurate and cross platform timezone calculations. It is used by Trigger's change management interface to allow for strict adherance to scheduled maintenance events.

Used by:

- `trigger.acl.db`
- `trigger.changemgmt`
- `trigger.netdevices`

SimpleParse

SimpleParse is an extremely fast parser generator for Python that converts EBNF grammars into parsers. It is used by Trigger's ACL parser to allow us to translate ACLs from flat files into vendor-agnostic objects.

Used by:

- `trigger.acl.parser`

Package tools

We strongly recommend using pip to install Trigger as it is newer and generally better than `easy_install`. In either case, these tools will automatically install of the dependencies for you quickly and easily.

Other Dependencies

This documentation is incomplete and is being improved.

Know for now that if you want to use the integrated load queue, you must have the Python MySQL bindings.

- python-mysql (MySQLdb)

3.1.2 Installing Trigger

The following steps will get you the very basic functionality and will be improved over time. As mentioned at the top of this document, if you have any feedback or questions, please get get in touch!

Install Trigger package

Using pip:

```
sudo pip install trigger
```

From source (which will use `easy_install`):

```
sudo python setup.py install
```

Create configuration directory

This can be customized using the `PREFIX` configuration variable within `settings.py` and defaults to `/etc/trigger`:

```
sudo mkdir /etc/trigger
```

Copy settings.py

Trigger expects `settings.py` to be in `/etc/trigger`:

```
sudo cp conf/trigger_settings.py /etc/trigger/settings.py
```

If you really don't like this, you may override the default location by setting the environment variable `TRIGGER_SETTINGS` to the desired location. If you go this route, you must make sure all Trigger-based tools have this set prior to any imports!

Copy autoacl.py

```
sudo cp conf/autoacl.py /etc/trigger/autoacl.py
```

If you're using a non-standard location, be sure to update the `AUTOACL_FILE` configuration variable within `settings.py` with the location of `autoacl.py`!

Copy netdevices.xml

```
sudo cp conf/netdevices.xml /etc/trigger/netdevices.xml
```

Create MySQL Database

Trigger currently (but hopefully not for too much longer) uses MySQL for the automated ACL load queue used by the `load_acl` and `acl` utilities. If you want to use these tools, you need to create a MySQL database and make sure you also have the Python `MySQLdb` module installed.

Find `conf/acl_queue_schema.sql` in the source distribution and import the `queue` and `acl_queue` tables into a database of your choice. It's probably best to create a unique database and database user for this purpose, but we'll leave that up to you.

Example import:

```
% mysql trigger -u trigger_user -p < ./conf/acl_queue_schema.sql
```

3.1.3 Verifyng Functionality

Once the dependencies are installed, try doing stuff.

NetDevices

Try instantiating NetDevices, which holds your device metadata:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> dev = nd.find('test1-abc.net.aol.com')
```

ACL Parser

Try parsing an ACL using the ACL parser (the `tests` directory can be found within the Trigger distribution):

```
>>> from trigger.acl import parse
>>> acl = parse(open("tests/data/acl.test"))
>>> len(acl.terms)
103
```

ACL Database

Try loading the AclsDB to inspect automatic associations. First directly from autoacl:

```
>>> from trigger.acl.autoacl import autoacl
>>> autoacl(dev)
set(['juniper-router.policer', 'juniper-router-protect'])
```

And then inherited from autoacl by AclsDB:

```
>>> from trigger.acl.db import AclsDB
>>> a = AclsDB()
>>> a.get_acl_set(dev)
>>> dev.implicit_acls
set(['juniper-router.policer', 'juniper-router-protect'])
```

CHAPTER

FOUR

CONFIGURATION

This is a work in progress, but it's not a bad start. Please have a look and give us feedback on how we can improve!

4.1 Configuration and defaults

This document describes the configuration options available.

If you're using the default loader, you must create or copy the provided `trigger_settings.py` module and make sure it is in `/etc/trigger/settings.py` on the local system.

4.1.1 A Word about Defaults

There are two Trigger components that rely on Python modules to be provided on disk in `/etc/trigger` and these are:

- `trigger.acl.autoacl` at `/etc/trigger/autoacl.py`
- `trigger.conf` at `/etc/trigger/settings.py`

If your custom configuration either cannot be found or fails to import, Trigger will fallback to the defaults.

settings.py

Using a custom settings.py

You may override the default location using the `TRIGGER_SETTINGS` environment variable.

For example, set this variable and fire up the Python interpreter:

```
% export TRIGGER_SETTINGS=/home/jathan/sandbox/trigger/conf/trigger_settings.py
% python
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.environ.get('TRIGGER_SETTINGS')
'/home/j/jathan/sandbox/netops/trigger/conf/trigger_settings.py'
>>> from trigger.conf import settings
```

Observe that it doesn't complain. You have loaded `settings.py` from a custom location!

Using global defaults

If you don't want to specify your own `settings.py`, it will warn you and fallback to the defaults:

```
>>> from trigger.conf import settings
trigger/conf/__init__.py:114: RuntimeWarning: Module could not be imported from /tmp/trigger/settings
  warnings.warn(str(err) + ' Using default global settings.', RuntimeWarning)
```

autoacl()

The `trigger.netdevices` and `trigger.acl` modules require `autoacl()`.

Trigger wants to import the `autoacl()` function from either a module you specify or, failing that, the default location.

Using a custom autoacl()

You may override the default location of the module containing the autoacl() function using the `AUTOACL_FILE` environment variable just like how you specified a custom location for `settings.py`.

Using default autoacl()

Just as with `settings.py`, the same goes for `autoacl()`:

```
>>> from trigger.acl.autoacl import autoacl
trigger/acl/autoacl.py:44: RuntimeWarning: Function autoacl() could not be found in /etc/trigger/auto
  warnings.warn(msg, RuntimeWarning)
```

Keep in mind this `autoacl()` has the expected signature but does nothing with the arguments and only returns an empty set:

```
>>> autoacl('foo')
set([])
```

4.1.2 Configuration Directives

Global settings

PREFIX

This is where Trigger should look for its essential files including `autoacl.py` and `netdevices.xml`.

Default:

```
'/etc/trigger'
```

USE_GPG_AUTH

Toggles whether or not we should use GPG authentication for storing TACACS credentials in the user's `.tacacsrc` file. Set to `False` to use the old .tackf encryptoin method, which sucks but requires almost no overhead. Should be `False` unless instructions/integration is ready for GPG. At this time the documentation for the GPG support is incomplete.

Default:

```
False
```

TACACSRC_KEYFILE

Only used if GPG auth is disabled. This is the location of the file that contains the passphrase used for the two-way hashing of the user credentials within the `.tacacsrc` file.

Default:

```
'/etc/trigger/.tackf'
```

DEFAULT_REALM

Default login realm to store user credentials (username, password) for general use within the `.tacacsrc` file.

Default:

```
'aol'
```

FIREWALL_DIR

Location of firewall policy files.

Default:

```
'/data/firewalls'
```

TFTPROOT_DIR

Location of the tftproot directory.

Default:

```
'/data/tftproot'
```

INTERNAL_NETWORKS

A list of `IPy.IP` objects describing your internally owned networks. All network blocsk owned/operated and considered a part of your network should be included. The defaults are private IPv4 networks defined by RFC 1918.

Default:

```
[IPy.IP("10.0.0.0/8"), IPy.IP("172.16.0.0/12"), IPy.IP("192.168.0.0/16")]
```

SUCCESS_EMAILS

A list of email addresses to email when things go well (such as from `load_acl --auto`).

Default:

```
[]
```

FAILURE_EMAILS

A list of email addresses to email when things go not well.

Default:

```
[]
```

Twister settings

These settings are used to customize the timeouts and methods used by Trigger to connect to network devices.

DEFAULT_TIMEOUT

Default timeout in seconds for commands executed during a session. If a response is not received within this window, the connection is terminated.

Default:

```
300
```

TELNET_TIMEOUT

Default timeout in seconds for initial telnet connections.

Default:

```
60
```

SSH_TYPES

A list of manufacturers that support SSH logins. Only add one if ALL devices of that manufacturer have SSH logins enabled. (Don't forget the trailing comma when you add a new entry.)

Default:

```
['ARISTA NETWORKS', 'CITRIX', 'JUNIPER', 'NETSCREEN TECHNOLOGIES']
```

VALID_VENDORS

A tuple of strings containing the names of valid manufacturer names. These are currently defaulted to what Trigger supports internally. Do not modify this unless you know what you're doing!

Default:

```
('ARISTA NETWORKS', 'CISCO SYSTEMS', 'DELL', 'JUNIPER', 'FOUNDRY', 'CITRIX', 'BROCADE')
```

IOSLIKE_VENDORS

A tuple of strings containing the names of vendors that basically just emulate Cisco's IOS and can be treated accordingly for the sake of interaction.

Default:

```
('ARISTA NETWORKS', 'BROCADE' 'CISCO SYSTEMS', 'DELL', 'FOUNDRY')
```

NetDevices settings

AUTOACL_FILE

Path to the explicit module file for autoacl.py so that we can still perform `from trigger.acl.autoacl import autoacl` without modifying `sys.path`.

Default:

```
'/etc/trigger/autoacl.py'
```

NETDEVICES_FORMAT

One of `xml`, `json`, `sqlite`. This MUST match the actual format of `NETDEVICES_FILE` or it won't work for obvious reasons.

You may override this location by setting the `NETDEVICES_FORMAT` environment variable to the format of the file.

Default:

```
'xml'
```

NETDEVICES_FILE

Path to netdevices device metadata source file, which is used to populate `NetDevices`. This may be JSON, XML, or a SQLite3 database. You must set `NETDEVICES_FORMAT` to match the type of data.

You may override this location by setting the `NETDEVICES_FILE` environment variable to the path of the file.

Default:

```
'/etc/trigger/netdevices.xml'
```

VALID_OWNERS

A tuple of strings containing the names of valid owning teams for `NetDevice` objects. This is intended to be a master list of the valid owners to have a central configuration entry to easily reference. Please see the sample settings file for an example to use in your environment.

Default:

```
()
```

Redis settings

REDIS_HOST

Redis master server. This will be used unless it is unreachable.

Default:

```
'127.0.0.1'
```

REDIS_PORT

The Redis port.

Default:

```
6379
```

REDIS_DB

The Redis DB to use.

Default:

```
0
```

Database settings

These will eventually be replaced with Redis or another task queue solution (such as Celery). For now, you'll need to populate this with information for your MySQL database.

These are all self-explanatory, I hope.

DATABASE_NAME

The name of the database.

Default:

```
''
```

DATABASE_USER

The username to use to connect to the database.

Default:

```
''
```

DATABASE_PASSWORD

The password for the user account used to connect to the database.

Default:

```
''
```

DATABASE_HOST

The host on which your MySQL databse resides.

Default:

```
'127.0.0.1'
```

DATABASE_PORT

The destination port used by MySQL.

Default:

```
3306
```

Access-list Management settings

These are various settings that control what files may be modified, by various tools and libraries within the Trigger suite. These settings are specific to the functionality found within the `trigger.acl` module.

IGNORED_ACLS

This is a list of FILTER names of ACLs that should be skipped or ignored by tools. These should be the names of the filters as they appear on devices. We want this to be mutable so it can be modified at runtime.

Default:

```
[]
```

NONMOD_ACLS

This is a list of FILE names of ACLs that shall not be modified by tools. These should be the names of the files as they exist in `FIREWALL_DIR`. Trigger expects ACLs to be prefixed with `'acl.'`.

Default:

```
[]
```

VIPS

This is a dictionary mapping of real IP to external NAT IP address for used by your connecting host(s) (aka jump host). This is used primarily by `load_acl` in the event that a connection from a real IP fails (such as via tftp) or when explicitly passing the `--no-vip` flag. Format: `{local_ip:  external_ip}`

Default:

```
{}
```

Access-list loading & rate-limiting settings

All of the following esttings are currently only used by `load_acl`. If and when the `load_acl` functionality gets moved into the library API, this may change.

AUTOLOAD_FILTER

A list of FILTER names (not filenames) that will be skipped during automated loads (`load_acl --auto`). This setting was renamed from `AUTOLOAD_BLACKLIST`; usage of that name is being phased out.

Default:

```
[]
```

AUTOLOAD_FILTER_THRESH

A dictionary mapping for FILTER names (not filenames) and a numeric threshold. Modify this if you want to create a list that if over the specified number of devices will be treated as bulk loads.

For now, we provided examples so that this has more context/meaning. The current implementation is kind of broken and doesn't scale for data centers with a large of number of devices.

Default:

```
{}
```

AUTOLOAD_BULK_THRESH

Any ACL applied on a number of devices >= this number will be treated as bulk loads. For example, if this is set to 5, any ACL applied to 5 or more devices will be considered a bulk ACL load.

Default:

```
10
```

BULK_MAX_HITS

This is a dictionary mapping of filter names to the number of bulk hits. Use this to override `BULK_MAX_HITS_DEFAULT`. Please note that this number is used PER EXECUTION of `load_acl --auto`. For example if you ran it once per hour, and your bounce window were 3 hours, this number should be the total number of expected devices per ACL within that allotted bounce window. Yes this is confusing and needs to be redesigned.)

Examples: + 1 per load_acl execution; ~3 per day, per 3-hour bounce window + 2 per load_acl execution; ~6 per day, per 3-hour bounce window

Default:

BULK_MAX_HITS_DEFAULT

If an ACL is bulk but not defined in `BULK_MAX_HITS`, use this number as max_hits. For example using the default value of 1, that means load on one device per ACL, per data center or site location, per `load_acl --auto` execution.

Default:

```
1
```

On-Call Engineer Display settings

GET_CURRENT_ONCALL

This variable should reference a function that returns data for your on-call engineer, or failing that `None`. The function should return a dictionary that looks like this:

```
{
    'username': 'mrengineer',
    'name': 'Joe Engineer',
    'email': 'joe.engineer@example.notreal'
}
```

Default:

```
lambda x=None: x
```

CM Ticket Creation settings

CREATE_CM_TICKET

This variable should reference a function that creates a CM ticket and returns the ticket number, or `None`. It defaults to `_create_cm_ticket_stub`, which can be found within the `settings.py` source code and is a simple function that takes any arguments and returns `None`.

Default:

```
_create_cm_ticket_stub
```

CHAPTER

FIVE

DOCUMENTATION

Please note that all documentation is written with users of Python 2.6 in mind. It's safe to assume that Trigger will not work properly on Python versions earlier than Python 2.6.

For now, most of our documentation is automatically generated form the source code documentation, which is usually very detailed. As we move along, this will change, especially with regards to some of the more creative ways in which we use Trigger's major functionality.

5.1 API Documentation

Trigger's core API is made up of several components.

5.1.1 `trigger.acl` — ACL parsing library

Trigger's ACL parser.

This library contains various modules that allow for parsing, manipulation, and management of network access control lists (ACLs). It will parse a complete ACL and return an ACL object that can be easily translated to any supported vendor syntax.

`trigger.acl.`**`parse`**(*input_data*)

Parse a complete ACL and return an ACL object. This should be the only external interface to the parser.

Parameters data – An ACL policy as a string or file-like object.

class `trigger.acl.`**`ACL`**(*name=None*, *terms=None*, *format=None*, *family=None*)

An abstract access-list object intended to be created by the `parse()` function.

`name_terms`()

Assign names to all unnamed terms.

`output`(*format=None*, **largs*, ***kwargs*)

Output the ACL data in the specified format.

`output_ios`(*replace=False*)

Output the ACL in IOS traditional format.

Parameters replace – If set the ACL is preceded by a `no access-list` line.

`output_ios_brocade`(*replace=False*, *receive_acl=False*)

Output the ACL in Brocade-flavored IOS format.

The difference between this and "traditional" IOS are:

- Stripping of comments
- Appending of `ip rebind-acl` or `ip rebind-receive-acl` line

Parameters

- **replace** – If set the ACL is preceded by a `no access-list` line.
- **receive_acl** – If set the ACL is suffixed with a `ip rebind-receive-acl' instead of ``ip rebind-acl`.

`output_ios_named`(*replace=False*)
Output the ACL in IOS named format.

Parameters **replace** – If set the ACL is preceded by a `no access-list` line.

`output_iosxr`(*replace=False*)
Output the ACL in IOS XR format.

Parameters **replace** – If set the ACL is preceded by a `no ipv4 access-list` line.

`output_junos`(*replace=False*, *family=None*)
Output the ACL in JunOS format.

Parameters

- **replace** – If set the ACL is wrapped in a `firewall { replace: ... }` section.
- **family** – If set, the value is used to wrap the ACL in a `family inet { ...}` section.

`strip_comments`()
Strips all comments from ACL header and all terms.

`trigger.acl.autoacl`

This module controls when ACLs get auto-applied to network devices, in addition to what is specified in acls.db.

This is primarily used by `AclsDB` to populate the **implicit** ACL-to-device mappings.

No changes should be made to this module. You must specify the path to the autoacl logic inside of `settings.py` as `AUTOACL_FILE`. This will be exported as `autoacl` so that the module path for the `autoacl()` function will still be `trigger.autoacl.autoacl()`.

This trickery allows us to keep the business-logic for how ACLs are mapped to devices out of the Trigger packaging.

If you do not specify a location for `AUTOACL_FILE` or the module cannot be loaded, then a default `autoacl()` function ill be used.

`trigger.acl.autoacl.`**`autoacl`**(*dev*, *explicit_acls=None*)
Given a NetDevice object, returns a set of **implicit** (auto) ACLs. We require a device object so that we don't have circular dependencies between netdevices and autoacl.

This function MUST return a `set()` of acl names or you will break the ACL associations. An empty set is fine, but it must be a set!

Parameters

- **dev** – A `NetDevice` object.
- **explicit_acls** – A set containing names of ACLs. Default: set()

```
>>> dev = nd.find('test1-abc')
>>> dev.manufacturer
JUNIPER
>>> autoacl(dev)
set(['juniper-router-protect', 'juniper-router.policer'])
```

NOTE: If the default function is returned it does nothing with the arguments and always returns an empty set.

trigger.acl.db

Redis-based replacement of the legacy acls.db file. This is used for interfacing with the explicit and automatic ACL-to-device mappings.

```
>>> from trigger.netdevices import NetDevices
>>> from trigger.acl.db import AclsDB
>>> nd = NetDevices()
>>> dev = nd.find('test1-abc')
>>> a = AclsDB()
>>> a.get_acl_set(dev)
set(['juniper-router.policer', 'juniper-router-protect', 'abc123'])
>>> a.get_acl_set(dev, 'explicit')
set(['abc123'])
>>> a.get_acl_set(dev, 'implicit')
set(['juniper-router.policer', 'juniper-router-protect'])
>>> a.get_acl_dict(dev)
{'all': set(['abc123', 'juniper-router-protect', 'juniper-router.policer']),
 'explicit': set(['abc123']),
  'implicit': set(['juniper-router-protect', 'juniper-router.policer'])}
```

`trigger.acl.db.`**`get_matching_acls`**(*wanted*, *exact=True*, *match_acl=True*, *match_device=False*, *nd=None*)

Return a sorted list of the names of devices that have at least one of the wanted ACLs, and the ACLs that matched on each. Without 'exact', match ACL name by startswith.

To get a list of devices, matching the ACLs specified:

```
>>> adb.get_matching_acls(['abc123'])
[('fw1-xyz.net.aol.com', ['abc123']), ('test1-abc.net.aol.com', ['abc123'])]
```

To get a list of ACLS matching the devices specified using an explicit match (default) by setting match_device=True:

```
>>> adb.get_matching_acls(['test1-abc'], match_device=True)
[]
>>> adb.get_matching_acls(['test1-abc.net.aol.com'], match_device=True)
[('test1-abc.net.aol.com', ['abc123', 'juniper-router-protect',
'juniper-router.policer'])]
```

To get a list of ACLS matching the devices specified using a partial match. Not how it returns all devices starting with 'test1-mtc':

```
>>> adb.get_matching_acls(['test1-abc'], match_device=True, exact=False)
[('test1-abc.net.aol.com', ['abc123', 'juniper-router-protect',
'juniper-router.policer'])]
```

`trigger.acl.db.`**`get_all_acls`**(*nd=None*)

Returns a dict keyed by acl names whose containing a set of NetDevices objects to which each acl is applied.

@nd can be your own NetDevices object if one is not supplied already

```
>>> all_acls = get_all_acls()
>>> all_acls['abc123']
set([<NetDevice: test1-abc.net.aol.com>, <NetDevice: fw1-xyz.net.aol.com>])
```

`trigger.acl.db.`**`get_bulk_acls`**(*nd=None*)
Returns a set of acls with an applied count over settings.AUTOLOAD_BULK_THRESH.

`trigger.acl.db.`**`populate_bulk_acls`**(*nd=None*)
Given a NetDevices instance, Adds bulk_acls attribute to NetDevice objects.

class `trigger.acl.db.`**`AclsDB`**
Container for ACL operations.

add/remove operations are for explicit associations only.

`add_acl`(*device*, *acl*)
Add explicit acl to device

```
>>> dev = nd.find('test1-mtc')
>>> a.add_acl(dev, 'acb123')
'added acl abc123 to test1-mtc.net.aol.com'
```

`get_acl_dict`(*device*)
Returns a dict of acl mappings for a @device, which is expected to be a NetDevice object.

```
>>> a.get_acl_dict(dev)
{'all': set(['115j', 'protectRE', 'protectRE.policer', 'test-bluej',
'testgreenj', 'testops_blockmj']),
'explicit': set(['test-bluej', 'testgreenj', 'testops_blockmj']),
'implicit': set(['115j', 'protectRE', 'protectRE.policer'])}
```

`get_acl_set`(*device*, *acl_set='all'*)
Return an acl set matching @acl_set for a given device. Match can be one of ['all', 'explicit', 'implicit']. Defaults to 'all'.

```
>>> a.get_acl_set(dev)
set(['testops_blockmj', 'testgreenj', '115j', 'protectRE',
'protectRE.policer', 'test-bluej'])
>>> a.get_acl_set(dev, 'explicit')
set(['testops_blockmj', 'test-bluej', 'testgreenj'])
>>> a.get_acl_set(dev, 'implicit')
set(['protectRE', 'protectRE.policer', '115j'])
```

`remove_acl`(*device*, *acl*)
Remove explicit acl from device.

```
>>> a.remove_acl(dev, 'acb123')
'removed acl abc123 from test1-mtc.net.aol.com'
```

`trigger.acl.parser`

Parse and manipulate network access control lists.

This library doesn't completely follow the border of the valid/invalid ACL set, which is determined by multiple vendors and not completely documented by any of them. We could asymptotically approach that with an enormous amount of testing, although it would require a 'flavor' flag (vendor, router model, software version) for full support. The realistic goal is to catch all the errors that we see in practice, and to accept all the ACLs that we use in practice, rather than to try to reject *every* invalid ACL and accept *every* valid ACL.

`trigger.acl.parser.`**`parse`**(*input_data*)
Parse a complete ACL and return an ACL object. This should be the only external interface to the parser.

Parameters data – An ACL policy as a string or file-like object.

class `trigger.acl.parser.`**`Comment`**(*data*)
Container for inline comments.

`output_ios`()
Output the Comment to IOS traditional format.

`output_ios_named`()
Output the Comment to IOS named format.

`output_iosxr`()
Output the Comment to IOS XR format.

`output_junos`()
Output the Comment to JunOS format.

class `trigger.acl.parser.`**`Term`**(*name=None*, *action='accept'*, *match=None*, *modifiers=None*, *inactive=False*, *isglobal=False*, *extra=None*)
An individual term from which an ACL is made

`output` (*format*, **largs*, ***kwargs*)
Output the term to the specified format

Parameters format – The desired output format.

`output_ios`(*prefix='‘*)
Output term to IOS traditional format.

`output_ios_named`(*prefix='‘*)
Output term to IOS named format.

`output_iosxr`(*prefix='‘*)
Output term to IOS XR format.

`output_junos`()
Convert the term to JunOS format.

`set_action_or_modifier`(*action*)
Add or replace a modifier, or set the primary action. This method exists for the convenience of parsers.

class `trigger.acl.parser.`**`Protocol`**(*arg*)
A protocol object used for access membership tests in `Term` objects. Acts like an integer, but stringify into a name if possible.

class `trigger.acl.parser.`**`ACL`**(*name=None*, *terms=None*, *format=None*, *family=None*)
An abstract access-list object intended to be created by the `parse()` function.

`name_terms`()
Assign names to all unnamed terms.

`output` (*format=None*, **largs*, ***kwargs*)
Output the ACL data in the specified format.

`output_ios`(*replace=False*)
Output the ACL in IOS traditional format.

Parameters replace – If set the ACL is preceded by a `no access-list` line.

`output_ios_brocade`(*replace=False*, *receive_acl=False*)
Output the ACL in Brocade-flavored IOS format.

The difference between this and "traditional" IOS are:

- Stripping of comments
- Appending of `ip rebind-acl` or `ip rebind-receive-acl` line

Parameters

- **replace** – If set the ACL is preceded by a `no access-list` line.
- **receive_acl** – If set the ACL is suffixed with a `ip rebind-receive-acl' instead of ``ip rebind-acl`.

output_ios_named(*replace=False*)
Output the ACL in IOS named format.

Parameters **replace** – If set the ACL is preceded by a `no access-list` line.

output_iosxr(*replace=False*)
Output the ACL in IOS XR format.

Parameters **replace** – If set the ACL is preceded by a `no ipv4 access-list` line.

output_junos(*replace=False*, *family=None*)
Output the ACL in JunOS format.

Parameters

- **replace** – If set the ACL is wrapped in a `firewall { replace: ... }` section.
- **family** – If set, the value is used to wrap the ACL in a `family inet { ...}` section.

strip_comments()
Strips all comments from ACL header and all terms.

`trigger.acl.parser.`**literals**(*d*)
Longest match of all the strings that are keys of 'd'.

`trigger.acl.parser.`**IP**(*arg*)
Wrapper for IPy.IP to intercept exception text and make it more user-friendly.

class `trigger.acl.parser.`**Policer**(*name*, *data*)
Container class for policer policy definitions. This is a dummy class for now, that just passes it through as a string.

class `trigger.acl.parser.`**PolicerGroup**(*format=None*)
Container for Policer objects. Juniper only.

`trigger.acl.parser.`**S**(*prod*)
Wrap your grammar token in this to call your helper function with a list of each parsed subtag, instead of the raw text. This is useful for performing modifiers.

Parameters **prod** – The parser product.

exception `trigger.acl.parser.`**ParseError**(*reason*, *line=None*, *column=None*)
Error parsing/normalizing an ACL that tries to tell you where it failed

trigger.acl.exceptions

All exceptions for trigger.acl. None of these have docstrings. We're working on it!

exception `trigger.acl.exceptions.`**ParseError**(*reason*, *line=None*, *column=None*)
Error parsing/normalizing an ACL that tries to tell you where it failed

`trigger.acl.queue`

Database interface for automated ACL queue. Used primarily by `load_acl` and `acl`` commands for manipulating the work queue.

```
>>> from trigger.acl.queue import Queue
>>> q = Queue()
>>> q.list()
(('dc1-abc.net.aol.com', 'datacenter-protect'), ('dc2-abc.net.aol.com',
'datacenter-protect'))
```

class `trigger.acl.queue.`**`Queue`**(*verbose=True*)

Interacts with firewalls database to insert/remove items into the queue. You may optionally suppress informational messages by passing `verbose=False` to the constructor.

Parameters **verbose** (*Boolean*) – Toggle verbosity

`complete`(*device*, *acls*)

Integrated queue only.

Mark a device and associated ACLs as complete my updating loaded to current timestampe. Migrated from clear_load_queue() in load_acl.

`delete`(*acl*, *routers=None*, *escalation=False*)

Delete an ACL from the firewall database queue.

Attempts to delete from integrated queue. If ACL test fails, then item is deleted from manual queue.

`insert`(*acl*, *routers*, *escalation=False*)

Insert an ACL and associated devices into the ACL load queue.

Attempts to insert into integrated queue. If ACL test fails, then item is inserted into manual queue.

`list`(*queue='integrated'*, *escalation=False*)

List items in the queue, defauls to integrated queue.

Valid queue arguments are 'integrated' or 'manual'.

`remove`(*acl*, *routers*, *escalation=False*)

Integrated queue only.

Mark an ACL and associated devices as "removed" (loaded=0). Intended for use when performing manual actions on the load queue when troubleshooting or addressing errors with automated loads. This leaves the items in the database but removes them from the active queue.

`trigger.acl.tools`

Various tools for use in scripts or other modules. Heavy lifting from tools that have matured over time have been moved into this module.

`trigger.acl.tools.`**`create_trigger_term`**(*source_ips=[]*, *dest_ips=[]*, *source_ports=[]*, *dest_ports=[]*, *protocols=[]*, *action=['accept']*, *name='generated_term'*)

Constructs & returns a Term object from constituent parts.

`trigger.acl.tools.`**`create_access`**(*terms_to_check*, *new_term*)

Breaks a new_term up into separate constituent parts so that they can be compared in a check_access test.

Returns a list of terms that should be inserted.

`trigger.acl.tools.`**`check_access`**(*terms_to_check*, *new_term*, *quiet=True*, *format='junos'*)

Determine whether access is permitted by a given ACL (list of terms).

Tests a new term against a list of terms. Return True if access in new term is permitted, or False if not.

Optionally displays the terms that apply and what edits are needed.

class `trigger.acl.tools.`**`ACLScript`**(*acl=None*, *mode='insert'*, *cmd='acl_script'*, *show_mods=True*, *no_worklog=False*, *no_changes=False*)

Interface to generating or modifying access-lists. Intended for use in creating command-line utilities using the ACL API.

`trigger.acl.tools.`**`process_bulk_loads`**(*work*, *max_hits=1*, *force_bulk=False*)

Formerly "process –ones".

Processes work dict and determines tuple of (prefix, site) for each device. Stores tuple as a dict key in prefix_hits. If prefix_hits[(prefix, site)] is greater than max_hits, remove all further matching devices from work dict.

By default if a device has no acls flagged as bulk_acls, it is not removed from the work dict.

Example:

- Device 'foo1-xyz.example.com' returns ('foo', 'xyz') as tuple.
- This is stored as prefix_hits[('foo', 'xyz')] = 1
- All further devices matching that tuple increment the hits for that tuple
- Any devices matching hit counter exceeds max_hits is removed from work dict

You may override max_hits to increase the num. of devices on which to load a bulk acl. You may pass force_bulk=True to treat all loads as bulk loads.

`trigger.acl.tools.`**`get_bulk_acls`**()

Returns a dict of acls with an applied count over settings.AUTOLOAD_BULK_THRESH

`trigger.acl.tools.`**`get_comment_matches`**(*aclobj*, *requests*)

Given an ACL object and a list of ticket numbers return a list of matching comments.

`trigger.acl.tools.`**`write_tmpacl`**(*acl*, *process_name='_tmpacl'*)

Write a temporary file to disk from an Trigger acl.ACL object & return the filename

`trigger.acl.tools.`**`diff_files`**(*old*, *new*)

Return a unified diff between two files

`trigger.acl.tools.`**`worklog`**(*title*, *diff*, *log_string='updated by express-gen'*)

Save a diff to the ACL worklog

5.1.2 `trigger.changemgmt` — Change management library

Abstract interface to bounce windows and moratoria.

class `trigger.changemgmt.`**`BounceStatus`**(*str*)

Class for bounce window statuses.

Objects stringify to 'red', 'green', or 'yellow', and can be compared against those strings. Objects can also be compared against each other. 'red' > 'yellow' > 'green'.

class `trigger.changemgmt.`**`BounceWindow`**(*status_by_hour*)

Build a bounce window based on a list of 24 BounceStatus objects.

Although the query API is generic and could accomodate any sort of bounce window policy, this constructor knows only about AOL's bounce windows, which operate on US Eastern time (worldwide), always change on hour boundaries, and are the same every day. If that ever changes, only this class will need to be updated.

End-users are not expected to create new BounceWindow objects; instead, use site_bounce() or NetDevice.site.bounce to get an object, then query its methods.

next_ok (*status*, *when=None*)
Return the next time at or after the specified time (default now) that it the bounce status will be at equal to or less than the given status. For example, next_ok('yellow') will return the time that the bounce window becomes yellow or green. Returns UTC time.

status (*when=None*)
Return a BounceStatus object for the specified time, or for now.

trigger.changemgmt.**site_bounce** (*site*, *oncallid=None*)
Return the bounce window for the given site.

5.1.3 trigger.cmds — Command execution library

Abstracts the execution of commands on network devices. Allows for integrated parsing and manipulation of return data for rapid integration to existing or newly created tools.

Commando superclass is intended to be subclassed. More documentation soon!

class trigger.cmds.**Commando** (*devices=None*, *max_conns=10*, *verbose=False*, *timeout=30*, *production_only=True*)
I run commands on devices but am not much use unless you subclass me and configure vendor-specific parse/generate methods.

arista_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

brocade_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

dell_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

foundry_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

generate_arista_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

generate_brocade_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

generate_dell_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

generate_foundry_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

generate_ios_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

generate_junos_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

generate_netscaler_cmd (*dev=None*)
Generate commands to be run on a device. If you don't overload this, it returns an empty list.

ios_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

junos_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

netscaler_parse (*data*, *device*)
Parse output from a device. Overload this to customize this default behavior.

run ()
Nothing happens until you execute this to perform the actual work.

set_data (*device*, *data*)
Another method for storing results. If you'd rather just change the default method for storing results, overload this. All default parse/generate methods call this.

class trigger.cmds.**NetACLInfo** (***args*)
Class to fetch and parse interface information. Exposes a config attribute which is a dictionary of devices passed to the constructor and their interface information.

Each device is a dictionary of interfaces. Each interface field will default to an empty list if not populated after parsing. Below is a skeleton of the basic config, with expected fields:

```
config {
    'device1': {
        'interface1': {
            'acl_in': [],
            'acl_out': [],
            'addr': [],
            'description': [],
            'subnets': [],
        }
    }
}
```

Interface field descriptions:

```
:addr: List of IPy.IP objects of interface addresses
:acl_in: List of inbound ACL names
:acl_out: List of outbound ACL names
:description: List of interface description(s)
:subnets: List of IPy.IP objects of interface networks/CIDRs
```

Example:

```
>>> n = NetACLInfo(devices=['jm10-cc101-lab.lab.aol.net'])
>>> n.run()
Fetching jm10-cc101-lab.lab.aol.net
>>> n.config.keys()
[<NetDevice: jm10-cc101-lab.lab.aol.net>]
>>> dev = n.config.keys()[0]
>>> n.config[dev].keys()
['lo0.0', 'ge-0/0/0.0', 'ge-0/2/0.0', 'ge-0/1/0.0', 'fxp0.0']
>>> n.config[dev]['lo0.0'].keys()
['acl_in', 'subnets', 'addr', 'acl_out', 'description']
>>> lo0 = n.config[dev]['lo0.0']
>>> lo0['acl_in']; lo0['addr']
['abc123']
[IP('66.185.128.160')]
```

IPsubnet (*addr*)
Given '172.20.1.4/24', return IP('172.20.1.0/24').

arista_parse (*data*, *device*)
Parse IOS config based on EBNF grammar

brocade_parse (*data*, *device*)
Parse IOS config based on EBNF grammar

foundry_parse (*data*, *device*)
Parse IOS config based on EBNF grammar

generate_arista_cmd (*dev*)
Similar to IOS, but:

- Arista has now "show conf" so we have to do "show run"
- The regex used in the CLI for Arista is more "precise" so we have to change the pattern a little bit compared to the on in generate_ios_cmd

generate_brocade_cmd (*dev*)
This is the "show me all interface information" command we pass to IOS devices

generate_foundry_cmd (*dev*)
This is the "show me all interface information" command we pass to IOS devices

generate_ios_cmd (*dev*)
This is the "show me all interface information" command we pass to IOS devices

generate_junos_cmd (*dev*)
Generates an etree.Element object suitable for use with JunoScript

ios_parse (*data*, *device*)
Parse IOS config based on EBNF grammar

ipv4_cidr_to_netmask (*bits*)
Convert CIDR bits to netmask

junos_parse (*data*, *device*)
Do all the magic to parse Junos interfaces

5.1.4 `trigger.conf` — Configuration & Settings module

Settings and configuration for Trigger.

Values will be read from the module specified by the `TRIGGER_SETTINGS` environment variable, and then from trigger.conf.global_settings; see the global settings file for a list of all possible variables.

If `TRIGGER_SETTINGS` is not set, it will attempt to load from `/etc/trigger/settings.py` and complains if it can't. The primary public interface for this module is the `settings` variable, which is a module object containing the variables found in `settings.py`.

```
>>> from trigger.conf import settings
>>> settings.FIREWALL_DIR
'/data/firewalls'
>>> settings.REDIS_HOST
'127.0.0.1'
```

class `trigger.conf.`**`DummySettings`**
Emulates settings and returns empty strings on attribute gets.

`trigger.conf.`**`import_path`** (*full_path*, *global_name*)
Import a file with full path specification. Allows one to import from anywhere, something `__import__` does not do.

Also adds the module to `sys.modules` as module_name

Parameters

- **full_path** – The absolute path to the module .py file
- **global_name** – The name assigned to the module in sys.modules. To avoid confusion, the global_name should be the same as the variable to which you're assigning the returned module.

Returns a module object.

class `trigger.conf.`**`BaseSettings`**

Common logic for settings whether set by a module or by the user.

5.1.5 `trigger.netdevices` — Network device metadata library

The heart and soul of Trigger, NetDevices is an abstract interface to network device metadata and ACL associations.

Parses netdevices.xml and makes available a dictionary of `NetDevice` objects, which is keyed by the FQDN of every network device.

Other interfaces are non-public.

Example:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> dev = nd['test1-abc.net.aol.com']
>>> dev.manufacturer, dev.make
('JUNIPER', 'MX960-BASE-AC')
>>> dev.bounce.next_ok('green')
datetime.datetime(2010, 4, 9, 9, 0, tzinfo=<UTC>)
```

`trigger.netdevices.`**`device_match`**(*name*, *production_only=True*)

Return a matching `NetDevice` object based on partial name. Return `None` if no match or if multiple matches is cancelled:

```
>>> device_match('test')
2 possible matches found for 'test':
  [ 1] test1-abc.net.aol.com
  [ 2] test2-abc.net.aol.com
  [ 0] Exit

Enter a device number: 2
<NetDevice: test2-abc.net.aol.com>
```

If there is only a single match, that device object is returned without a prompt:

```
>>> device_match('fw')
Matched 'fw1-xyz.net.aol.com'.
<NetDevice: fw1-xyz.net.aol.com>
```

class `trigger.netdevices.`**`NetDevice`**(*data=None*)

Almost all the attributes are populated by netdevices._populate() and are mostly dependent upon the source data. This is prone to implementation problems and should be revisited in the long-run as there are certain fields that are baked into the core functionality of Trigger.

Users usually won't create `NetDevice` objects directly! Rely instead upon `NetDevices` to do this for you.

allowable(*action*, *when=None*)
Ok to perform the specified action? Returns a boolean value. False means a bounce window conflict. For now 'load-acl' is the only valid action and moratorium status is not checked.

dump()
Prints details for a device.

is_firewall()
Am I a firewall?

is_netscaler()
Am I a NetScaler?

is_router()
Am I a router?

is_switch()
Am I a switch?

next_ok(*action*, *when=None*)
Return the next time at or after the specified time (default now) that it will be ok to perform the specified action.

class trigger.netdevices.**NetDevices**(*production_only=True*)
Returns an immutable Singleton dictionary of `NetDevice` objects. By default it will only return devices for which `adminStatus=='PRODUCTION'`.

There are hardly any use cases where `NON-PRODUCTION` devices are needed, and it can cause real bugs of two sorts:

1. trying to contact unreachable devices and reporting spurious failures,
2. hot spares with the same nodeName.

You may override this by passing `production_only=False`.

class _actual(*production_only=True*)
This is the real class that stays active upon instantiation. All attributes are inherited by NetDevices from this object. This means you do NOT reference `_actual` itself, and instead call the methods from the parent object.

Right:

```
>>> nd = NetDevices()
>>> nd.search('fw')
[<NetDevice: fw1-xyz.net.aol.com>]
```

Wrong:

```
>>> nd._actual.search('fw')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method match() must be called with _actual
instance as first argument (got str instance instead)
```

all()
Returns all NetDevice objects.

find(*key*)
Return either the exact nodename, or a unique dot-delimited prefix. For example, if there is a node 'test1-abc.net.aol.com', then any of find('test1-abc') or find('test1-abc.net') or find('test1-abc.net.aol.com') will match, but not find('test1').

Parameters key (*string*) – Hostname prefix to find.

Returns NetDevice object

get_devices_by_type(*devtype*)
Returns a list of NetDevice objects with deviceType matching type.

Known deviceTypes: ['FIREWALL', 'ROUTER', 'SWITCH', 'DWDM']

list_firewalls()
Returns a list of NetDevice objects with deviceType of FIREWALL

list_routers()
Returns a list of NetDevice objects with deviceType of ROUTER

list_switches()
Returns a list of NetDevice objects with deviceType of SWITCH

match(***kwargs*)
Attempt to match values to all keys in @kwargs by dynamically building a list comprehension. Will throw errors if the keys don't match legit NetDevice attributes.

Keys and values are case IN-senstitive. Matches against non-string values will FAIL.

Example by reference:

```
>>> nd = NetDevices()
>>> myargs = {'onCallName':'Data Center', 'model':'FCSLB'}
>>> mydevices = nd(**kwargs)
```

Example by keyword arguments:

```
>>> mydevices = nd(oncallname='data center', model='fcslb')
```

Returns List of NetDevice objects

search(*token, field='nodeName'*)
Returns a list of NetDevice objects where other is in `dev.nodeName`. The getattr call in the search will allow a `AttributeError` from a bogus field lookup so that you don't get an empty list thinking you performed a legit query.

For example, this:

```
>>> field = 'bacon'
>>> [x for x in nd.all() if 'ash' in getattr(x, field)]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'NetDevice' object has no attribute 'bacon'
```

Is better than this:

```
>>> [x for x in nd.all() if 'ash' in getattr(x, field, '')]
[]
```

Because then you know that 'bacon' isn't a field you can search on.

Parameters
- **token** (*string*) – Token to search match on in @field
- **field** (*string*) – The field to match on when searching

Returns List of NetDevice objects

5.1.6 `trigger.netscreen` — Juniper NetScreen firewall parser

Parses and manipulates firewall policy for Juniper NetScreen firewall devices. Broken apart from acl.parser because the approaches are vastly different from each other.

class `trigger.netscreen.`**`NSAddress`**(*name=None*, *zone=None*, *addr=None*, *comment=None*)
Container for individual address items.

class `trigger.netscreen.`**`NSAddressBook`**(*name='ANY'*, *zone=None*)
Container for address book entries.

class `trigger.netscreen.`**`NSGroup`**(*name=None*, *group_type='address'*, *zone=None*)
Container for address/service groups.

class `trigger.netscreen.`**`NSPolicy`**(*name=None*, *address_book=<trigger.netscreen.NSAddressBook object at 0x4f31650>*, *service_book=<trigger.netscreen.NSServiceBook object at 0x4f317d0>*, *address_groups=[ ]*, *service_groups=[ ]*, *source_zone='Untrust'*, *destination_zone='Trust'*, *id=0*, *action='permit'*, *isglobal=False*)
Container for individual policy definitions.

class `trigger.netscreen.`**`NSRawGroup`**(*data*)
Container for group definitions.

class `trigger.netscreen.`**`NSRawPolicy`**(*data*, *isglobal=0*)
Container for policy definitions.

class `trigger.netscreen.`**`NSService`**(*name=None*, *protocol=None*, *source_port=(1*, *65535)*, *destination_port=(1*, *65535)*, *timeout=0*, *predefined=False*)
Container for individual service items.

class `trigger.netscreen.`**`NSServiceBook`**(*entries=[ ]*)
Container for built-in service entries and their defaults.

Example: service = NSService(name="stupid_http") service.set_source_port((1,65535)) service.set_destination_port(80) service.set_protocol('tcp') print service.output()

class `trigger.netscreen.`**`NetScreen`**
Parses and generates NetScreen firewall policy.

`handle_raw_netscreen`(*rows*)
The parser will hand it's final output to this function, which decodes and puts everything in the right place.

`netmask2cidr`(*ipstr*)
Converts dotted-quad netmask to cidr notation

`parse`(*data*)
Parse policy into list of NSPolicy objects.

`trigger.netscreen.`**`concatenate_grp`**(*x*)
Used by NetScreen class when grouping policy members.

5.1.7 `trigger.tacacsrc` — Network credentials library

Abstract interface to .tacacsrc credentials file.

Designed to interoperate with the legacy DeviceV2 implementation, but provide a reasonable API on top of that. The name and format of the .tacacsrc file are not ideal, but compatibility matters.

`trigger.tacacsrc.`**`get_device_password`**(*device=None*)

Fetch the password for a device/realm or create a new entry for it. If device is not passed, 'settings.DEFAULT_REALM is used, which is default realm for most devices.

Parameters device – Realm or device name to updated

`trigger.tacacsrc.`**`prompt_credentials`**(*device*, *user=None*)

Prompt for username, password and return them as Credentials namedtuple.

Parameters

- **device** – Device or realm name to store
- **user** – (Optional) If set, use as default username

`trigger.tacacsrc.`**`convert_tacacsrc`**()

Converts old .tacacsrc to new .tacacsrc.gpg.

`trigger.tacacsrc.`**`update_credentials`**(*device*, *username=None*)

Update the credentials for a given device/realm. Assumes the same username that is already cached unless it is passed.

This may seem redundant at first compared to Tacacsrc.update_creds() but we need this factored out so that we don't end up with a race condition when credentials are messed up.

Returns True if it actually updated something or None if it didn't.

Parameters

- **device** – Device or realm name to update
- **username** – Username for credentials

class `trigger.tacacsrc.`**`Tacacsrc`**(*tacacsrc_file=None*, *use_gpg=False*, *generate_new=False*)

Encrypts, decrypts and returns credentials for use by network devices and other tools.

Pass use_gpg=True to force GPG, otherwise it relies on settings.USE_GPG_AUTH

`*_old` functions should be removed after everyone is moved to the new system.

`update_creds`(*creds*, *realm*, *user=None*)

Update username/password for a realm/device and set self.creds_updated bit to trigger .write().

Parameters

- **creds** – Dictionary of credentials keyed by realm
- **realm** – The realm to update within the creds dict
- **user** – (Optional) Username passed to prompt_credentials()

`user_has_gpg`()

Checks if user has .gnupg directory and .tacacsrc.gpg file.

`write`()

Writes .tacacsrc(.gpg) using the accurate method (old vs. new).

5.1.8 `trigger.twister` — Asynchronous device interaction library

Login and basic command-line interaction support using the Twisted asynchronous I/O framework. The Trigger Twister is just like the Mersenne Twister, except not at all.

class `trigger.twister.`**`IncrementalXMLTreeBuilder`**(*callback*, **args*, ***kwargs*)
Version of XMLTreeBuilder that runs a callback on each tag.

We need this because JunoScript treats the entire session as one XML document. IETF NETCONF fixes that.

class `trigger.twister.`**`Interactor`**
Creates an interactive shell. Intended for use as an action with pty_connect(). See gong for an example.

`connectionMade`()
Fire up stdin/stdout once we connect.

`dataReceived`(*data*)
And write data to the terminal.

class `trigger.twister.`**`IoslikeSendExpect`**(*dev*, *commands*, *incremental=None*, *with_errors=False*, *timeout=None*, *command_interval=0*)
Action for use with TriggerTelnet. Take a list of commands, and send them to the device until we run out or one errors. Wait for a prompt after each.

`connectionMade`()
Do this when we connect.

`dataReceived`(*bytes*)
Do this when we get data.

`timeoutConnection`()
Do this when we timeout.

class `trigger.twister.`**`TriggerClientFactory`**(*deferred*, *creds=None*, *init_commands=None*)
Factory for all clients. Subclass me.

`clientConnectionFailed`(*connector*, *reason*)
Do this when the connection fails.

`clientConnectionLost`(*connector*, *reason*)
Do this when the connection is lost.

class `trigger.twister.`**`TriggerSSHChannelBase`**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
Base class for SSH Channels. setup_channelOpen() should be called by channelOpen() in the child class.

`loseConnection`()
Terminate the connection. Link this to the transport method of the same name.

`setup_channelOpen`(*data*)
Call me in your subclass in self.channelOpen():

```
def channelOpen(self, data):
    self.setup_channelOpen(data)
    self.conn.sendRequest(self, 'shell', '')
    # etc.
```

`timeoutConnection`()
Do this when the connection times out.

class `trigger.twister.`**`TriggerSSHChannelFactory`**(*deferred*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=None*, *channel=None*, *command_interval=0*)
Intended to be used as a parent of automated SSH channels (e.g. Junoscript, NetScreen, NetScaler) to eliminate boiler plate in those subclasses.

class `trigger.twister.`**`TriggerSSHConnection`**
Used to manage, you know, an SSH connection.

`channelClosed`(*channel*)
Close the channel when we're done.

`serviceStarted`()
Open the channel once we start.

class `trigger.twister.`**`TriggerSSHJunoscriptChannel`**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
Run Junoscript commands on a Juniper router. This completely assumes that we are the only channel in the factory (a TriggerJunoscriptFactory) and walks all the way back up to the factory for its arguments.

`channelOpen`(*data*)
Do this when channel opens.

`dataReceived`(*data*)
Do this when we receive data.

class `trigger.twister.`**`TriggerSSHNetscalerChannel`**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
Same as TriggerSSHJunoscriptChannel but for NetScreen. Mostly copy-pasted, and probably needs refactoring but it works.

`channelOpen`(*data*)
Do this when channel opens.

`dataReceived`(*bytes*)
Do this when we receive data.

class `trigger.twister.`**`TriggerSSHNetscreenChannel`**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
An SSH Channel to interact with NetScreens (running ScreenOS).

`channelOpen`(*data*)
Do this when the channel opens.

`dataReceived`(*bytes*)
Do this when we receive data.

class `trigger.twister.`**`TriggerSSHPtyChannel`**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
Used by pty_connect() to turn up an SSH pty channel.

`channelOpen`(*data*)
Setup the terminal when the channel opens.

class `trigger.twister.`**`TriggerSSHPtyClientFactory`**(*deferred*, *action*, *creds=None*, *display_banner=None*, *init_commands=None*)
Factory for an interactive SSH connection. 'action' is a Protocol that will be connected to the session after login. Use it to interact with the user and pass along commands.

class `trigger.twister.`**`TriggerSSHTransport`**
Call with magic factory attributes 'creds', a tuple of login credentials, and 'channel', the class of channel to open.

connectionSecure()
Once we're secure, authenticate.

receiveError(*reason*, *desc*)
Do this when we receive an error.

sendDisconnect(*reason*, *desc*)
Trigger disconnect of the transport.

verifyHostKey(*pubKey*, *fingerprint*)
Verify host key, but don't actually verify. Awesome.

class trigger.twister.**TriggerSSHUserAuth**(*user*, *instance*)
Perform user authentication over SSH.

getGenericAnswers(*name*, *information*, *prompts*)
Send along the password when authentication mechanism is not 'password'. This is most commonly the case with 'keyboard-interactive', which even when configured within self.preferredOrder, does not work using default getPassword() method.

getPassword(*prompt=None*)
Send along the password.

ssh_USERAUTH_BANNER(*packet*)
Display SSH banner.

ssh_USERAUTH_FAILURE(*packet*)
An almost exact duplicate of SSHUserAuthClient.ssh_USERAUTH_FAILURE modified to forcefully disconnect. If we receive authentication failures, instead of looping until the server boots us and performing a sendDisconnect(), we raise a LoginFailure and call loseConnection().

See the base docstring for the method signature.

class trigger.twister.**TriggerTelnet**(*timeout=60*)
Telnet-based session. Primarily used by IOS-like type devices.

enableRemote(*option*)
Allow telnet clients to enable options if for some reason they aren't enabled already (e.g. ECHO). (Ref: http://bit.ly/wkFZFg) For some reason Arista Networks hardware is the only vendor that needs this method right now.

login_state_machine(*bytes*)
Track user login state.

state_enable()
Special Foundry breakage because they don't do auto-enable from TACACS by default. Use 'aaa authentication login privilege-mode'. Also, why no space after the Password: prompt here?

state_enable_pw()
Pass the enable password from the factory or NetDevices

state_logged_in()
Once we're logged in, exit state machine and pass control to the action.

state_login_pw()
Pass the login password from the factory or NetDevices

state_password()
After we got password prompt, check for enabled prompt.

state_percent_error()
Found a % error message. Don't return immediately because we don't have the error text yet.

state_raise_error()
Do this when we get a login failure.

state_username()
After we've gotten username, check for password prompt.

timeoutConnection()
Do this when we timeout logging in.

class trigger.twister.**TriggerTelnetClientFactory**(*deferred*, *action*, *creds=None*, *loginpw=None*, *enablepw=None*, *init_commands=None*)
Factory for a telnet connection.

trigger.twister.**execute_ioslike**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *loginpw=None*, *enablepw=None*, *command_interval=0*)
Connect to a Cisco/IOS-like device over telnet. See execute_junoscript().

trigger.twister.**execute_junoscript**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*)
Connect to a Juniper and enable XML mode. Sequentially execute all the XML commands in the iterable 'commands' (ElementTree.Element objects suitable for wrapping in <rpc>). Returns a deferred, whose callback will get a sequence of all the XML results after the connection is finished.

If any command returns xnm:error, the connection is dropped immediately and the errback will fire with the failed command; or, set 'with_errors' to get the exception objects in the list instead. Connection failures will still fire the errback.

Any None object in the command sequence will result in a None being placed in the output sequence, with no command issued to the router.

@incremental (optional) will be called with an empty sequence immediately on connecting, and each time a result comes back with the list of all results.

@commands is usually just a list. However, you can have also make it a generator, and have it and @incremental share a closure to some state variables. This allows you to determine what commands to execute dynamically based on the results of previous commands. This implementation is experimental and it might be a better idea to have the 'incremental' callback determine what command to execute next; it could then be a method of an object that keeps state.

> BEWARE: Your generator cannot block; you must immediately decide what next command to execute, if any.

@timeout is the command timeout in seconds or None to disable. The default is in settings.DEFAULT_TIMEOUT; CommandTimeout errors will result if a command seems to take longer than that to run. LoginTimeout errors are always possible and cannot be disabled.

trigger.twister.**execute_netscaler**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*)
Connect to a NetScaler device. See execute_junoscript().

trigger.twister.**execute_netscreen**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*)
Connect to a NetScreen device. See execute_junoscript().

trigger.twister.**has_ioslike_error**(*s*)
Test whether a string seems to contain an IOS error.

trigger.twister.**has_junoscript_error**(*tag*)
Test whether an Element contains a Junoscript xnm:error.

`trigger.twister.`**`has_netscaler_error`**(*s*)
Test whether a string seems to contain a NetScaler error.

`trigger.twister.`**`is_awaiting_confirmation`**(*prompt*)
Checks if a prompt is asking for us for confirmation and returns a Boolean.

Parameters **prompt** – The prompt string to check

`trigger.twister.`**`pty_connect`**(*device*, *action*, *creds=None*, *display_banner=None*, *ping_test=False*, *init_commands=None*)
Connect to a device and log in. Use SSHv2 or telnet as appropriate.

Parameters

- **device** – A `NetDevice` object.
- **action** – A Protocol object (not class) that will be activated when

the session is ready.

Parameters **creds** – is a 2-tuple (username, password). By default, .tacacsrc AOL

credentials will be used. Override that here.

Parameters **display_banner** – Will be called for SSH pre-authentication banners.

It will receive two args, 'banner' and 'language'. By default, nothing will be done with the banner.

Parameters **ping_test** – If set, the device is pinged and succeed in order to

proceed.

Parameters **init_commands** – A list of commands to execute upon logging into

the device.

5.1.9 `trigger.utils` — CLI tools and utilities library

A collection of CLI tools and utilities used by Trigger.

`trigger.utils.cli`

Command-line interface utilities for Trigger tools. Intended for re-usable pieces of code like user prompts, that don't fit in other utils modules.

`trigger.utils.cli.`**`yesno`**(*prompt*, *default=False*, *autoyes=False*)
Present a yes-or-no prompt, get input, and return a boolean.

Parameters

- **prompt** – Prompt text
- **default** – Yes if True; No if False
- **autoyes** – Automatically return True

`trigger.utils.cli.`**`get_terminal_width`**()
Find and return stdout's terminal width, if applicable.

`trigger.utils.cli.`**`get_terminal_size`**()
Find and return stdouts terminal size as (height, width)

class `trigger.utils.cli.`**`Whirlygig`**`(`*start_msg='‘, done_msg='‘, max=100*`)`

Prints a whirlygig for use in displaying pending operation in a command-line tool. Guaranteed to make the user feel warm and fuzzy and be 1000% bug-free.

Parameters

- **start_msg** – The status message displayed to the user (e.g. “Doing stuff:”)
- **done_msg** – The completion message displayed upon completion (e.g. “Done.”)
- **max** – Integer of the number of whirlygig repetitions to perform

Example:

```
>>> Whirly("Doing stuff:", "Done.", 12).run()
```

`run``()`

Executes the whirlygig!

class `trigger.utils.cli.`**`NullDevice`**

Used to supress output to sys.stdout.

Example:

print “1 - this will print to STDOUT” original_stdout = sys.stdout # keep a reference to STDOUT sys.stdout = NullDevice() # redirect the real STDOUT print “2 - this won’t print” sys.stdout = original_stdout # turn STDOUT back on print “3 - this will print to SDTDOUT”

`trigger.utils.cli.`**`print_severed_head`**`()`

Thanks to Jeff Sullivan for this best error message ever.

`trigger.utils.cli.`**`min_sec`**`(`*secs*`)`

Takes epoch timestamp and returns string of minutes:seconds.

`trigger.utils.cli.`**`pretty_time`**`(`*t*`)`

Print a pretty version of timestamp, including timezone info. Expects the incoming datetime object to have proper tzinfo.

```
>>> import datetime
>>> from pytz import timezone
>>> localzone = timezone('US/Eastern')
<DstTzInfo 'US/Eastern' EST-1 day, 19:00:00 STD>
>>> t = datetime.datetime.now(localzone)
>>> print t
2011-07-19 12:40:30.820920-04:00
>>> print pretty_time(t)
09:40 PDT
>>> t = datetime.datetime(2011,07,20,04,13,tzinfo=localzone)
>>> print t
2011-07-20 04:13:00-05:00
>>> print pretty_time(t)
tomorrow 02:13 PDT
```

`trigger.utils.cli.`**`proceed`**`()`

Present a proceed prompt. Return True if Y, else False.

trigger.utils.network

Functions that perform network-based things like ping, port tests, etc.

`trigger.utils.network.`**`ping`**`(`*host*, *count=1*, *timeout=5*`)`

Returns pass/fail for a ping. Supports POSIX only.

Parameters

- **host** – Hostname or address
- **count** – Repeat count
- **timeout** – Timeout in seconds

```
>>> from trigger.utils import network
>>> network.ping('aol.com')
True
>>> network.ping('192.168.199.253')
False
```

`trigger.utils.network.`**`test_tcp_port`**`(`*host*, *port=23*, *timeout=5*`)`

Attempts to connect to a TCP port. Returns a boolean.

Parameters

- **host** – Hostname or address
- **port** – Destination port
- **timeout** – Timeout in seconds

```
>>> network.test_tcp_port('aol.com', 80)
True
>>> network.test_tcp_port('aol.com', 12345)
False
```

`trigger.utils.network.`**`address_is_internal`**`(`*ip*`)`

Determines if an IP address is internal to your network. Relies on networks specified in `settings.INTERNAL_NETWORKS`.

Parameters **ip** – IP address to test.

```
>>> network.address_is_internal('1.1.1.1')
False
```

trigger.utils.rcs

Provides a CVS like wrapper for local RCS (Revision Control System) with common commands.

class `trigger.utils.rcs.`**`RCS`**`(`*filename*, *create=True*`)`

Simple wrapper for CLI `rcs` command. An instance is bound to a file.

Parameters

- **file** – The filename (or path) to use
- **create** – If set, create the file if it doesn't exist

```
>>> from trigger.utils.rcs import RCS
>>> rcs = RCS('foo')
>>> rcs.lock()
True
>>> f = open('foo', 'w')
>>> f.write('bar\n')
>>> f.close()
>>> rcs.checkin('This is my commit message')
```

```
True
>>> print rcs.log()
RCS file: RCS/foo,v
Working file: foo
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
----------------------------
revision 1.2
date: 2011/07/08 21:01:28;  author: jathan;  state: Exp;  lines: +1 -0
This is my commit message
----------------------------
revision 1.1
date: 2011/07/08 20:56:53;  author: jathan;  state: Exp;
first commit
```

checkin(*logmsg='none'*, *initial=False*, *verbose=False*)
Perform an RCS checkin. If successful this also unlocks the file, so there is no need to unlock it afterward.

Parameters

- **logmsg** – The RCS commit message
- **initial** – Initialize a new RCS file, but do not deposit any revision
- **verbose** – Print command output

```
>>> rcs.checkin('This is my commit message')
True
```

lock(*verbose=False*)
Perform an RCS checkout with lock. Returns boolean of whether lock was sucessful.

Parameters **verbose** – Print command output

```
>>> rcs.lock()
True
```

lock_loop(*callback=None*, *timeout=5*, *verbose=False*)
Keep trying to lock the file until a lock is obtained.

Parameters

- **callback** – The function to call after lock is complete
- **timeout** – How long to sleep between lock attempts
- **verbose** – Print command output

Default:

```
>>> rcs.lock_loop(timeout=1)
Sleeping to wait for the lock on the file: foo
Sleeping to wait for the lock on the file: foo
```

Verbose:

```
>>> rcs.lock_loop(timeout=1, verbose=True)
RCS/foo,v  -->  foo
co: RCS/foo,v: Revision 1.2 is already locked by joe.
Sleeping to wait for the lock on the file: foo
RCS/foo,v  -->  foo
co: RCS/foo,v: Revision 1.2 is already locked by joe.
```

log()
Returns the RCS log as a string (see above).

unlock(*verbose=False*)
Perform an RCS checkout with unlock (for cancelling changes).

Parameters **verbose** – Print command output

```
>>> rcs.unlock()
True
```

5.2 Tutorial

Coming Soon.

5.3 Usage Documentation

Once you've properly installed Trigger, you might want to know how to use it. Please have a look at the usage documentation!

5.3.1 Command-line Tools

Blah blah blah command-line stuff here.

The following tools are included:

acl - ACL database interface

About

acl is used to interface with the ACL database and queue. It is a simple command to manage or determine access-list associations, and allows you to inject or remove an ACL from the load queue.

Usage

Here is the usage output:

```
% acl
Usage: acl [options]

Options:
-h, --help            show this help message and exit
-s, --staged          list currently staged ACLs
-l, --list            list ACLs currently in integrated (automated) queue
-m, --listmanual      list entries currently in manual queue
-i, --inject          inject into load queue
-c, --clear           clear from load queue
-x, --exact           match entire name, not just start
-d, --device-name-only
                      don’t match on ACL
-a ADD, --add=ADD     add an acl to explicit ACL database, example: "acl -a
                      abc123 test1-abc test2-abc"
-r REMOVE, --remove=REMOVE
                      remove an acl from explicit ACL database, example:
                      "acl -r abc123 -r xyz246 test1-abc"
-q, --quiet           be quiet! (For use with scripts/cron)
```

Examples

Managing ACL associations

Adding an ACL association When adding an association, you must provide the full ACL name. You may, however, use the short name of any devices to which you’d like to associate that ACL:

```
% acl -a jathan-special test1-abc test2-abc
added acl jathan-special to test1-abc.net.aol.com
added acl jathan-special to test2-abc.net.aol.com
```

If you try to add an association for a device that does not exist, it will complain:

```
% acl -a foo godzilla-router
skipping godzilla-router: invalid device

Please use --help to find the right syntax.
```

Removing an ACL association Removing associations are subject to the same restrictions as additions, however in this example we’ve referenced the devices by FQDN:

```
% acl -r jathan-special test1-abc.net.aol.com test2-abc.net.aol.com
removed acl jathan-special from test1-abc.net.aol.com
removed acl jathan-special from test2-abc.net.aol.com
```

Confirm the removal and observe that it returns nothing:

```
% acl jathan-special
%
```

If you try to remove an ACL that is not associated, it will complain:

```
% acl -r foo test1-abc
test1-abc.net.aol.com does not have acl foo
```

Searching for an ACL or device You may search by full or partial names of ACLs or devices. When you search for results, ACLs are checked first. If there are no matches then device names are checked second. In either case, the pattern must match the beginning of the name of the ACL or device.

You may search for the exact name of the ACL we just added:

```
% acl jathan-special
test1-abc.net.aol.com                  jathan-special
test2-abc.net.aol.com                  jathan-special
```

A partial ACL name will get you the same results in this case:

```
% acl jathan
test1-abc.net.aol.com                  jathan-special
test2-abc.net.aol.com                  jathan-special
```

A partial name will return all matching objects with names starting with the pattern. Because there are no ACLs starting with `'test1'` matching devices are returned instead:

```
% acl test1
test1-abc.net.aol.com                  jathan-special abc123 xyz246
test1-def.net.aol.com                  8 9 10
test1-xyz.net.aol.com                  8 9 10
```

If you want to search for an exact ACL match, use the `-x` flag:

```
% acl -x jathan
No results for ['jathan']
```

Or if you want to match devices names only, use the `-d` flag:

```
% acl -d jathan-special
No results for ['jathan-special']
```

Working with the load queue Not finished yet...

Integrated queue

Manual queue

aclconv - ACL Converter

About

aclconv Convert an ACL on stdin, or a list of ACLs, from one format to another. Input format is determined automatically. Output format can be given with -f or with one of -i/-o/-j/-x. The name of the output ACL is determined automatically, or it can be specified with -n.

Usage

Here is the usage output:

```
Options:
-h, --help            show this help message and exit
-f FORMAT, --format=FORMAT
-o, --ios-named       Use IOS named ACL output format
-j, --junos           Use JunOS ACL output format
-i, --ios             Use IOS old-school ACL output format
-x, --iosxr           Use IOS XR ACL output format
-n ACLNAME, --name=ACLNAME
```

Examples

Coming Soon™.

go - Device connector

About

go Go connects to network devices and automatically logs you in using cached TACACS credentials. It supports telnet, SSHv1/v2.

PLEASE NOTE: go is still named **gong** (aka "Go NG") within the Trigger packaging due to legacy issues with naming conflicts. This will be changing in the near future.

Usage

Here is the usage output:

```
% gong
Usage: gong [options] [device]

Automatically log into network devices using cached TACACS credentials.
```

```
Options:
  -h, --help  show this help message and exit
  -o, --oob   Connect to device out of band first.
```

Examples

Caching credentials If you haven't cached your credentials, you'll be prompted to:

```
% gong test2-abc
Connecting to test2-abc.net.aol.com.  Use ^X to exit.
/home/jathan/.tacacsrc not found, generating a new one!

Updating credentials for device/realm 'tacacsrc'
Username: jathan
Password:
Password (again):

Fetching credentials from /home/jathan/.tacacsrc
test2-abc#
```

This functionality is provided by `Tacacsrc`.

Connecting to devices Using gong is pretty straightforward if you've already cached your credentials:

```
% gong test1-abc
Connecting to test1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
--- JUNOS 10.0S8.2 built 2010-09-07 19:55:32 UTC
jathan@test1-abc>
```

Full or partial hostname matches are also acceptable:

```
% gong test2-abc.net.aol.com
Connecting to test2-abc.net.aol.com.  Use ^X to exit.
```

If there are multiple matches, you get to choose:

```
% gong test1
3 possible matches found for 'test1':
 [ 1] test1-abc.net.aol.com
 [ 2] test1-def.net.aol.com
 [ 3] test1-xyz.net.aol.com
 [ 0] Exit

Enter a device number: 3
Connecting to test1-xyz.net.aol.com.  Use ^X to exit.
```

If a partial name only has a single match, it will connect automatically:

```
% gong test1-a
Matched 'test1-abc.net.aol.com'.
Connecting to test1-abc.net.aol.com.  Use ^X to exit.
```

Out-of-band support If a device has out-of-band (OOB) terminal server and ports specified within `NetDevices`, you may telnet to the console by using the `-o` flag:

```
% gong -o test2-abc
OOB Information for test2-abc.net.aol.com
telnet ts-abc.oob.aol.com 1234
Connecting you now...
Trying 10.302.134.584...
Connected to test2-abc.net.aol.com
Escape character is '^]'.


User Access Verification

Username:
```

Executing commands upon login You may create a `.gorc` file in your home directory, in which you may specify commands to be executed upon login to a device. The commands are specified by the vendor name. Here is an example:

```
; .gorc - Example file to show how .gorc would work

[init_commands]
; Specify the commands you would like run upon login for each vendor name. The
; vendor name must match the one found in the CMDB for the manufacturer of the
; hardware. Currently these are:
;
;  Arista: ARISTA NETWORKS
; Brocade: BROCADE
;   Cisco: CISCO SYSTEMS
;  Citrix: CITRIX
;    Dell: DELL
; Foundry: FOUNDRY
; Juniper: JUNIPER
;
; Format:
;
; VENDOR:
;     command1
;     command2
;
JUNIPER:
    request system reboot
    set cli timestamp
    monitor start messages
    show system users

CISCO SYSTEMS:
    term mon
    who

ARISTA NETWORKS:
    python-shell

FOUNDRY:
    show clock

BROCADE:
    show clock
```

(You may also find this file at `conf/gorc.example` within the Trigger source tree.)

Notice for **JUNIPER** one of the commands specified is `request system reboot`. This is bad! But don't worry, only a very limited subset of root commands are allowed to be specified within the `.gorc`, and these are:

```
get
monitor
ping
set
show
term
terminal
traceroute
who
whoami
```

Any root commands not permitted will be filtered out prior to passing them along to the device.

Here is an example of what happens when you connect to a `JUNIPER` device with the specified commands in the sample `.gorc` file displayed above:

```
% gong test1-abc
Connecting to test1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
--- JUNOS 10.0S8.2 built 2010-09-07 19:55:32 UTC
jathan@test1-abc> set cli timestamp
Mar 28 23:05:38
CLI timestamp set to: %b %d %T

jathan@test1-abc> monitor start messages

jathan@test1-abc> show system users
Jun 28 23:05:39
11:05PM  up 365 days, 13:44, 1 user, load averages: 0.09, 0.06, 0.02
USER     TTY      FROM                              LOGIN@  IDLE WHAT
jathan   p0       awesome.win.aol.com              11:05PM     - -cli (cli)

jathan@test1-abc>
```

netdev - CLI search for NetDevices

About

netdev is a command-line search interface for `NetDevices` metadata.

Usage

Here is the usage output:

```
% netdev
Usage: netdev [options]

Command-line search interface for 'netdevices.xml'.

Options:
  -h, --help            show this help message and exit
  -a, --acls            Search will return acls instead of devices.
  -l <DEVICE>, --list=<DEVICE>
                        List all information for individual DEVICE
  -s, --search          Perform a search based on matching criteria
  -L <LOCATION>, --location=<LOCATION>
                        For use with -s:  Match on site location.
  -n <NODENAME>, --nodename=<NODENAME>
                        For use with -s:  Match on full or partial nodeName.
                        NO REGEXP.
  -t <TYPE>, --type=<TYPE>
                        For use with -s:  Match on deviceType.  Must be
                        FIREWALL, ROUTER, or SWITCH.
  -o <OWNING TEAM NAME>, --owning-team=<OWNING TEAM NAME>
                        For use with -s:  Match on Owning Team (owningTeam).
  -O <ONCALL TEAM NAME>, --oncall-team=<ONCALL TEAM NAME>
                        For use with -s:  Match on Oncall Team (onCallName).
  -C <OWNING ORG>, --owning-org=<OWNING ORG>
                        For use with -s:  Match on cost center Owning Org.
                        (owner).
  -m <MANUFACTURER>, --manufacturer=<MANUFACTURER>
                        For use with -s:  Match on manufacturer.
  -b <BUDGET CODE>, --budget-code=<BUDGET CODE>
                        For use with -s:  Match on budget code
  -B <BUDGET NAME>, --budget-name=<BUDGET NAME>
                        For use with -s:  Match on budget name
  -k <MAKE>, --make=<MAKE>
                        For use with -s:  Match on make.
  -M <MODEL>, --model=<MODEL>
                        For use with -s:  Match on model.
  -N, --nonprod         Look for production and non-production devices.
```

Examples

Displaying an individual device You may use the `-l` option to list an individual device:

```
% netdev -l test1-abc

        Hostname:          test1-abc.net.aol.com
        Owning Org.:       12345678 - Network Engineering
        Owning Team:       Data Center
        OnCall Team:       Data Center

        Manufacturer:      JUNIPER
        Make:              M40 INTERNET BACKBONE ROUTER
        Model:             M40-B-AC
        Type:              ROUTER
        Location:          LAB CR10 16ZZ

        Project:           Test Lab
        Serial:            987654321
        Asset Tag:          0000012345
        Budget Code:        1234578 (Data Center)

        Admin Status:       PRODUCTION
        Lifecycle Status:   INSTALLED
        Operation Status:   MONITORED
        Last Updated:       2010-07-19 19:56:32.0
```

Partial names are also ok:

```
% netdev -l test1
3 possible matches found for 'test1':
 [ 1] test1-abc.net.aol.com
 [ 2] test1-def.net.aol.com
 [ 3] test1-xyz.net.aol.com
 [ 0] Exit

Enter a device number:
```

Searching by metadata To search you must specify the `-s` flag. All subsequent options are used as search terms. Each of the supported options coincides with attributes found on `NetDevice` objects.

You must provide at least one optional field or this happens:

```
% netdev -s
netdev: error: -s needs at least one other option, excluding -l.
```

Search for all Juniper switches in site "BBQ":

```
% netdev -s -t switch -m juniper -L bbq
```

All search arguments accept partial matches and are case-INsensitive, so this:

```
% netdev -s --manufacturer='CISCO SYSTEMS' --location=BBQ
```

Is equivalent to this:

```
% netdev -s --manufacturer=cisco --location=bbq
```

You can't mix `-l` (list) and `-s` (search) because they contradict each other:

```
% netdev -s -l -n test1
Usage: netdev [options]

netdev: error: -l and -s cannot be used together
```

5.3.2 Working with NetDevices

`NetDevices` is the core of Trigger's device interaction. Anything that communicates with devices relies on the metadata stored within `NetDevice` objects.

netdevices.xml

NetDevices reads in your `netdevices.xml` file that should be a dump of relevant metadata fields from your CMDB. If you don't have a CMDB, then you're going to have to populate this file manually. But you're a Python programmer, right? So you can come up with something spiffy!

Here is what the `netdevices.xml` file bundled with the Trigger source code looks like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Dummy version of netdevices.xml, with just one real entry modelded from the real file -->
<NetDevices>
    <device nodeName="test1-abc.net.aol.com">
        <adminStatus>PRODUCTION</adminStatus>
        <assetID>0000012345</assetID>
        <authMethod>tacacs</authMethod>
        <barcode>0101010101</barcode>
        <budgetCode>1234578</budgetCode>
        <budgetName>Data Center</budgetName>
        <coordinate>16ZZ</coordinate>
        <deviceType>ROUTER</deviceType>
        <enablePW>boguspassword</enablePW>
        <lastUpdate>2010-07-19 19:56:32.0</lastUpdate>
        <layer2>1</layer2>
        <layer3>1</layer3>
        <layer4>1</layer4>
        <lifecycleStatus>INSTALLED</lifecycleStatus>
        <loginPW></loginPW>
        <make>M40 INTERNET BACKBONE ROUTER</make>
        <manufacturer>JUNIPER</manufacturer>
        <model>M40-B-AC</model>
        <nodeName>test1-abc.net.aol.com</nodeName>
        <onCallEmail>nobody@aol.net</onCallEmail>
        <onCallID>17</onCallID>
        <onCallName>Data Center</onCallName>
        <owningTeam>Data Center</owningTeam>
        <OOBTerminalServerConnector>C</OOBTerminalServerConnector>
        <OOBTerminalServerFQDN>ts1.oob.aol.com</OOBTerminalServerFQDN>
        <OOBTerminalServerNodeName>ts1</OOBTerminalServerNodeName>
        <OOBTerminalServerPort>5</OOBTerminalServerPort>
        <OOBTerminalServerTCPPort>5005</OOBTerminalServerTCPPort>
        <operationStatus>MONITORED</operationStatus>
        <owner>12345678 - Network Engineering</owner>
        <projectName>Test Lab</projectName>
        <room>CR10</room>
        <serialNumber>987654321</serialNumber>
        <site>LAB</site>
    </device>
    ...
</NetDevices>
```

We plan to add support for different input sources including JSON, Sqlite, or "other" in the near future, but for now this is it.

By default the location of `netdevices.xml` can be specified one of two ways:

1. Specifing the path in the `NETDEVICESXML_FILE` environment variable, or;
2. modifying the value of of `settings.NETDEVICESXML_FILE`.

Getting Started

First things first, you must instantiate NetDevices. It has three things it requires before you can properly do this:

1. The `netdevices.xml` file must be readable and must properly parse (see above);
2. An instance of Redis.

3. The path to autoacl.py must be valid, and must properly parse.

How it works

The NetDevices object itself is an immutable, dictionary-like Singleton object. If you don't know what a Singleton is, it means that the actual there can only really only be one instance in any program. The actual instance object itself an instance of the inner _actual class which is stored in the module object as NetDevices._Singleton. This is done as a performance boost because many Trigger components require a NetDevices instance, and if we had to keep creating new ones, we'd be waiting forever each time one had to parse netdevices.xml all over again.

Upon startup, each <device> element found within netdevices.xml is used to create a NetDevice object. This object pulls in ACL associations from AclsDB.

The Singleton Pattern The NetDevices module object has a _Singleton attribute that defaults to None. Upon creating an instance, this is populated with the NetDevices._actual instance:

```
>>> nd = NetDevices()
>>> nd._Singleton
<trigger.netdevices._actual object at 0x2ae3dcf48710>
>>> NetDevices._Singleton
<trigger.netdevices._actual object at 0x2ae3dcf48710>
```

This is how new instances are prevented. Whenever you create a new reference by instantiating NetDevices again, what you are really doing is creating a reference to NetDevices._Singleton:

```
>>> other_nd = NetDevices()
>>> other_nd._Singleton
<trigger.netdevices._actual object at 0x2ae3dcf48710>
>>> nd._Singleton is other_nd._Singleton
True
```

The only time this would be an issue is if you needed to change the actual contents of your object (such as when debugging netdevices.xml or passing production_only=False). If you need to do this, set the value to None:

```
>>> NetDevices._Singleton = None
```

Then the next call to NetDevices() will start from scratch. Keep in mind because of this pattern it is not easy to have more than one instance (there are ways but we're not going to list them here!). All existing instances will inherit the value of NetDevices._Singleton:

```
>>> third_nd = NetDevices(production_only=False)
>>> third_nd._Singleton
<trigger.netdevices._actual object at 0x2ae3dcf506d0>
>>> nd._Singleton
<trigger.netdevices._actual object at 0x2ae3dcf506d0>
>>> third_nd._Singleton is nd._Singleton
True
```

Instantiating NetDevices

Throughout the Trigger code, the convention when instantiating and referencing a NetDevices instance, is to assign it to the variable nd. All examples will use this, so keep that in mind:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> len(nd)
3
```

By default, this only includes any devices for which `adminStatus` (aka administrative status) is `PRODUCTION`. This means that the device is used in your production environment. If you would like you get all devices regardless of `adminStatus`, you must pass `production_only=False` to the constructor:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices(production_only=False)
>>> len(nd)
4
```

The included sample `netdevices.xml` contains one device that is marked as `NON-PRODUCTION`.

What's in a NetDevice?

A `NetDevice` object has a number of attributes you can use creatively to correlate or identify them:

```
>>> dev = nd.find('test1-abc')
>>> dev
<NetDevice: test1-abc.net.aol.com>
```

Printing it displays the hostname:

```
>>> print dev
test1-abc.net.aol.com
```

You can dump the values:

```
>>> dev.dump()

        Hostname:          test1-abc.net.aol.com
        Owning Org.:       12345678 - Network Engineering
        Owning Team:       Data Center
        OnCall Team:       Data Center

        Manufacturer:      JUNIPER
        Make:              M40 INTERNET BACKBONE ROUTER
        Model:             M40-B-AC
        Type:              ROUTER
        Location:          LAB CR10 16ZZ

        Project:           Test Lab
        Serial:            987654321
        Asset Tag:         0000012345
        Budget Code:       1234578 (Data Center)

        Admin Status:      PRODUCTION
        Lifecycle Status:  INSTALLED
        Operation Status:  MONITORED
        Last Updated:      2010-07-19 19:56:32.0
```

You can reference them as attributes:

```
>>> dev.nodeName, dev.manufacturer, dev.deviceType
('test1-abc.net.aol.com', 'JUNIPER', 'ROUTER')
```

There are some special methods to perform identity tests:

```
>>> dev.is_router(), dev.is_switch(), dev.is_firewall()
(True, False, False)
```

You can view the ACLs assigned to the device:

```
>>> dev.explicit_acls
set(['abc123'])
>>> dev.implicit_acls
set(['juniper-router.policer', 'juniper-router-protect'])
>>> dev.acls
set(['juniper-router.policer', 'juniper-router-protect', 'abc123'])
```

Or get the next time it's ok to make changes to this device (more on this later):

```
>>> dev.bounce.next_ok('green')
datetime.datetime(2011, 7, 13, 9, 0, tzinfo=<UTC>)
>>> print dev.bounce.status()
red
```

Searching for devices

Like a dictionary

Since the object is like a dictionary, you may reference devices as keys by their hostnames:

```
>>> nd
{'test2-abc.net.aol.com': <NetDevice: test2-abc.net.aol.com>,
 'test1-abc.net.aol.com': <NetDevice: test1-abc.net.aol.com>,
 'lab1-switch.net.aol.com': <NetDevice: lab1-switch.net.aol.com>,
 'fw1-xyz.net.aol.com': <NetDevice: fw1-xyz.net.aol.com>}
>>> nd['test1-abc.net.aol.com']
<NetDevice: test1-abc.net.aol.com>
```

You may also perform any other operations to iterate devices as you would with a dictionary (`.keys()`, `.itervalues()`, etc.).

Special methods

There are a number of ways you can search for devices. In all cases, you are returned a list.

The simplest usage is just to list all devices:

```
>>> nd.all()
[<NetDevice: test2-abc.net.aol.com>, <NetDevice: test1-abc.net.aol.com>,
 <NetDevice: lab1-switch.net.aol.com>, <NetDevice: fw1-xyz.net.aol.com>]
```

Using `all()` is going to be very rare, as you're more likely to work with a subset of your devices.

Find a device by its shortname (minus the domain):

```
>>> nd.find('test1-abc')
<NetDevice: test1-abc.net.aol.com>
```

List devices by type (switches, routers, or firewalls):

```
>>> nd.list_routers()
[<NetDevice: test2-abc.net.aol.com>, <NetDevice: test1-abc.net.aol.com>]
>>> nd.list_switches()
[<NetDevice: lab1-switch.net.aol.com>]
>>> nd.list_firewalls()
[<NetDevice: fw1-xyz.net.aol.com>]
```

Perform a case-sensitive search on any field (it defaults to `nodeName`):

```
>>> nd.search('test')
[<NetDevice: test2-abc.net.aol.com>, <NetDevice: test1-abc.net.aol.com>]
>>> nd.search('test2')
[<NetDevice: test2-abc.net.aol.com>]
>>> nd.search('NON-PRODUCTION', 'adminStatus')
[<NetDevice: test2-abc.net.aol.com>]
```

Or you could just roll your own list comprehension to do the same thing:

```
>>> [d for d in nd.all() if d.adminStatus == 'NON-PRODUCTION']
[<NetDevice: test2-abc.net.aol.com>]
```

Perform a case-INsenstive search on any number of fields as keyword arguments:

```
>>> nd.match(oncallname='data center', adminstatus='non')
[<NetDevice: test2-abc.net.aol.com>]
>>> nd.match(manufacturer='netscreen')
[<NetDevice: fw1-xyz.net.aol.com>]
```

Helper function

Another nifty tool within the module is `device_match()`, which returns a NetDevice object:

```
>>> from trigger.netdevices import device_match
>>> device_match('test')
2 possible matches found for 'test':
 [ 1] test1-abc.net.aol.com
 [ 2] test2-abc.net.aol.com
 [ 0] Exit

Enter a device number: 2
<NetDevice: test2-abc.net.aol.com>
```

If there are multiple matches, it presents a prompt and lets you choose, otherwise it chooses for you:

```
>>> device_match('fw')
Matched 'fw1-xyz.net.aol.com'.
<NetDevice: fw1-xyz.net.aol.com>
```

5.3.3 Managing Credentials

`trigger.tacacsrc`

About

An abstract interface to .tacacsrc credentials file historically used by NEO developers for caching individual credentials. Supports GPG, which is the preferred method of credential storage, but backwards-compatible with DeviceV2.

Designed to interoperate with the legacy DeviceV2 implementation, but provide a reasonable API on top of that. The name and format of the .tacacsrc file are not ideal, but compatibility matters.

How it works

Detail tacacsrc process, integration

Usage

Migrate all documentation from here: http://wiki.office.aol.com/wiki/Trigger/Ops4#GPG_authentication_testing

5.4 FAQ

You guessed it: Coming Soon.

5.5 Change Log

Please review the `changelog`.

CHAPTER

SIX

CONTRIBUTING

Any hackers interested in improving Trigger (or even users interested in how Trigger is put together or released) please see the `development` page. It contains comprehensive info on contributing, repository layout, our release strategy, and more.

CHAPTER

SEVEN

GETTING HELP

If you've scoured the *Usage* and *API* documentation and still can't find an answer to your question, below are various support resources that should help. Please do at least skim the documentation before posting tickets or mailing list questions, however!

7.1 Mailing list

The best way to get help with using Trigger is via the trigger-users mailing list (Google Group). We'll do our best to reply promptly!

7.2 Twitter

Trigger has an official Twitter account, @pytrigger, which is used for announcements and occasional related news tidbits (e.g. "Hey, check out this neat article on Trigger!").

7.3 Email

If you don't do Twitter or mailing lists, please feel free to drop us an email at pytrigger@aol.com.

7.4 Bugs/ticket tracker

To file new bugs or search existing ones, please use the GitHub issue tracker, located at https://github.com/aol/trigger/issues.

7.5 IRC

IRC coming Soon™.

7.6 Wiki

We will use GitHub's built-in wiki located at https://github.com/aol/trigger/wiki.

CHAPTER

EIGHT

LICENSE

Trigger is licensed under the Clear BSD License which is based on the BSD 3-Clause License, and adds a term expressly stating it does not grant you any patent licenses.

For the explicit details, please see the `license` page.

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX