Trigger Documentation

Release 1.2.4

Jathan McCollum, Eileen Tschetter, Mark Ellzey Thomas, Michael

January 07, 2013

CONTENTS

"Keep your eyes on the prize, and your finger on the trigger."

CHAPTER

ONE

WHAT IS TRIGGER?

Trigger is a robust network engineering toolkit written in Python that was designed for interfacing with network devices and managing network configuration and security policy. It increases the speed and efficiency of managing large-scale networks while reducing the risk of human error.

CHAPTER

TWO

KEY FEATURES

Trigger is designed to work at scale and can support hundreds or thousands of network devices with ease. Here are some of things that make Trigger tick:

- Support for SSH, Telnet, and Juniper's Junoscript XML API
- Easily get an interactive shell or execute commands asynchronously.
- Leverage advanced event-driven functionality to manage any number of jobs in parallel and handle output or errors as they return.
- Powerful metadata interface for performing complex queries to group and associate network devices by name, manufacturer, type, location, and more.
- Encrypted storage of login credentials so you can interact without constantly being prompted to enter your password.
- Flexible access-list & firewall policy parser that can test access if access is permitted, or easily convert ACLs from one format to another.
- Detailed support for timezones and maintenance windows.
- A suite of tools for simplifying many common tasks.

New in version 1.2.

- Import your metadata from an existing RANCID installation to get up-and-running quickly!

CHAPTER

THREE

EXAMPLES

To illustrate how Trigger works, here are some basic examples of leveraging the API.

For these examples to work you must have already *installed* and *configured* Trigger, so if you haven't already please do that first!

3.1 Simple Examples

3.1.1 Working with metadata

Get a count of all your devices:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> len(nd)
5539
```

(Whoa! That's a lot!) Let's look up a device.

```
>>> dev = nd.find('edge1-abc')
>>> dev.vendor, dev.deviceType
(<Vendor: Juniper>, 'ROUTER')
>>> dev.has_ssh()
True
```

3.1.2 Get an interactive shell

Since this device has SSH, let's connect to it:

```
>>> dev = nd.find('edge1-abc')
>>> dev.connect()
Connecting to edge1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
--- JUNOS 10.2S6.3 built 2011-01-22 20:06:27 UTC
jathan@edge1-abc>
```

3.1.3 Work with access-lists

Let's start with a simple Cisco ACL:

```
>>> from trigger.acl import parse
>>> aclobj = parse("""access-list 123 permit tcp any host 10.20.30.40 eq 80""")
>>> aclobj.terms
[<Term: None>]
```

And convert it to Juniper format:

```
>>> aclobj.name_terms() # Juniper policy terms must have names
>>> aclobj.terms
[<Term: T1>]
>>> print '\n'.join(aclobj.output(format='junos'))
filter 123 {
    term T1 {
        from {
            destination-address {
                10.20.30.40/32;
            }
            protocol tcp;
            destination-port 80;
        }
        then {
            accept;
        }
    }
}
```

3.1.4 Cache your login credentials

Trigger will encrypt and store your credentials in a file called `.tacacsrc` in your home directory. We already had them cached in the previous examples, so I removed it and then:

```
>>> from trigger.tacacsrc import Tacacsrc
>>> tcrc = Tacacsrc()
/home/jathan/.tacacsrc not found, generating a new one!

Updating credentials for device/realm 'tacacsrc'
Username: jathan
Password:
Password (again):
>>> tcrc.creds['aol']
Credentials(username='jathan', password='boguspassword', realm='tacacsrc')
```

Passwords can be cached by realm. By default this realm is `'aol'`, but you can change that in the settings. Your credentials are encrypted and decrypted using a shared key. A more secure experimental GPG-encrypted method is in the works.

3.1.5 Login to a device using the `gong` script

Trigger includes a simple tool for end-users to connect to devices called `gong`. (It should be just `go` but we're in the future, so...):

```
$ gong foo1-cisco
Connecting to foo1-cisco.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
foo1-cisco#
```

```
foo1-cisco#show clock
20:52:05.777 UTC Sat Jun 23 2012
foo1-cisco#
```

Partial hostnames are supported, too:

```
$ gong foo1
2 possible matches found for 'foo1':
[ 1] foo1-abc.net.aol.com
[ 2] foo1-xyz.net.aol.com
[ 0] Exit

Enter a device number: 2
Connecting to foo1-xyz.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
foo1-xyz#
```

3.2 Slightly Advanced Examples

3.2.1 Execute commands asynchronously using Twisted

This is a little more advanced... so we saved it for last.

Trigger uses Twisted, which is a callback-based event loop. Wherever possible Twisted's implementation details are abstracted away, but the power is there for those who choose to wield it. Here's a super simplified example of how this might be accomplished:

```
from trigger.netdevices import NetDevices
from twisted.internet import reactor

nd = NetDevices()
dev = nd.find('foo1-abc')

def print_result(data):
    """Display results from a command"""
    print 'Result:', data

def stop_reactor(data):
    """Stop the event loop"""
    print 'Stopping reactor'
    if reactor.running:
        reactor.stop()

# Create an event chain that will execute a given list of commands on this
# device
async = dev.execute(['show clock'])

# When we get results from the commands executed, call this
async.addCallback(print_result)

# Once we're out of commands, or we an encounter an error, call this
async.addBoth(stop_reactor)

# Start the event loop
reactor.run()
```

Which outputs:

```
Result: ['21:27:46.435 UTC Sat Jun 23 2012\n']
Stopping reactor
```

Observe, however, that this only communicated with a single device.

3.2.2 Execute commands asynchronously using the Commando API

`Commando` tries to hide Twisted's implementation details so you don't have to deal with callbacks, while also implementing a worker pool so that you may easily communicate with multiple devices in parallel.

This is a base class that is intended to be extended to perform the operations you desire. Here is a basic example of how we might perform the same example above using `Commando` instead, but also communicating with a second device in parallel:

```
from trigger.cmds import Commando

class ShowClock(Commando):
    """Execute 'show clock' on a list of Cisco devices."""
    vendors = ['cisco']
    commands = ['show clock']

if __name__ == '__main__':
    device_list = ['foo1-abc.net.aol.com', 'foo2-xyz.net.aol.com']
    showclock = ShowClock(devices=device_list)
    showclock.run() # Commando exposes this to start the event loop

    print '\nResults:'
    print showclock.results
```

Which outputs:

```
Sending ['show clock'] to foo2-xyz.net.aol.com
Sending ['show clock'] to foo1-abc.net.aol.com
Received ['21:56:44.701 UTC Sat Jun 23 2012\n'] from foo2-xyz.net.aol.com
Received ['21:56:44.704 UTC Sat Jun 23 2012\n'] from foo1-abc.net.aol.com

Results:
{
    'foo1-abc.net.aol.com': {
        'show clock': '21:56:44.704 UTC Sat Jun 23 2012\n'
    },
    'foo2-xyz.net.aol.com': {
        'show clock': '21:56:44.701 UTC Sat Jun 23 2012\n'
    }
}
```

CHAPTER

FOUR

SUPPORTED PLATFORMS

Trigger currently officially supports devices manufactured by the following vendors:

- A10 Networks
  - All AX series application delivery controllers and server load balancers
- Arista Networks
  - All 7000-family switch platforms
- Brocade Networks
  - MLX routers
  - VDX switches
- Citrix Systems
  - NetScaler application delivery controllers and server load balancers
- Cisco Systems
  - All router and switch platforms running IOS
- 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)

It's worth noting that other vendors may actually work with the current libraries, but they have not been tested. The mapping of supported platforms is specified in `settings.py` as `SUPPORTED_PLATFORMS`. Modify it at your own risk!

CHAPTER

FIVE

GETTING STARTED

5.1 Before you begin

You might be required to tinker with some Python code. Don't worry, we'll be gentle!

Important: Throughout this documentation you will see commands or code preceded by a triple greater-than prompt (>>>). This indicates that they are being entered into a Python interpreter in *interactive mode*.

To start Python in *interactive mode*, it's as simple as executing `python` from a command prompt:

```
% python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

For more information, please see the official Python documentation on interactive mode.

5.2 Installation

Stable releases of Trigger are best installed using `pip` or `easy_install`; or you may download compressed source archives from any of the official locations. Detailed instructions and links may be found on the *Installation* page.

Please keep in mind that before you can truly use Trigger, you must configure it. This is not overly difficult, but it is an important step.

5.3 Configuration

To configure Trigger please see *Configuration and defaults*. Initial configuration is relatively easy. If you have any doubts, just start by using the defaults that are provided in the instructions so you can start tinkering.

To take full advantage of all of the features, there are some hurdles you have to jump through, but we are working on greatly simplifying this! 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!

CHAPTER

SIX

DOCUMENTATION

Please note that all documentation is written with users of Python 2.6 or higher 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.

6.1 Changelog

6.1.1 1.2.4

- The commands required to commit/save the configuration on a device are now attached to `NetDevice` objects under the `commit_commands` attribute, to make it easier to execute these commands without having to determine them for yourself.
- #56: Added a way to optionally perform a `commit full` operation on Juniper devices by defining a dictionary of attributes and values for matching devices using `JUNIPER_FULL_COMMIT_FIELDS`. This modifies the `commit_commands` that are assigned when the `NetDevice` object is created.
- #33: Console paging is now disabled by default for async SSH channels.

6.1.2 1.2.3

- #47: Added parsing of ranges for `fragment-offset` statements in Juniper ACLs.
- #49: Changed ACL parser to omit src/dst ports if port range is `0-65535`.
- #50: Fix typo that was causing Cisco parsing to generate an unhandled exception within `NetACLInfo`.
- Minor bugfix when checking device names and printing a warning within `Commando`.
- Updated docs to say we're using a interactive Python interpreter and added OpenHatch profile to contact info.

6.1.3 1.2.2

- #16: Arista support was added to `bin/load_acl`
- #45: Added "SSH-1.99" as a valid SSHv2 version in `test_ssh()` to fix a bug in which devices presenting this banner were errantly falling back to telnet and causing weird behavior during interactive sessions.

- #46: Changed `connect()` to pass the vendor name to `get_init_commands()` so that it is more explicit when debugging.
- #29: Added an extensible event notification system
  - A new pluggable notification system has been added in `notifications`, which defaults to email notifications. New event handlers and event types can be easily added and specified with the configuration using `NOTIFICATION_HANDLERS`.
  - The following changes have been made to `bin/load_acl`:
    - All alerts are now using the new notification system
    - `email_users()` moved to `send_email()`
    - All calls to send failures now call `send_notification()`
    - All calls to send successes now calls `send_email()`
  - In support of the new notification system, the following config settings have been added:
    - `EMAIL_SENDER` - The default email sender
    - `NOTIFICATION_SENDER` - The default notification sender
    - `SUCCESS_RECIPIENTS` - Hosts/addresses to send successes
    - `FAILURE_RECIPIENTS` - Hosts/addresses to send failures
    - `NOTIFICATION_HANDLERS` - A list of handler functions to process in order
  - A new utility module has been added to import modules in `importlib`, and `trigger.conf.import_path()` was moved to `import_module_from_path()` to bring these import tools under one roof.

6.1.4 1.2.1

- #30: Bugfix in `bin/acl` where tftproot was hard-coded. It now reads from `TFTPROOT_DIR`.
- #37: Fixed misleading "make discard" output from `bin/check_access`, to use the `Term.extra` attribute to store a user-friendly comment to make it clear that the term's action has been modified by the "make discard" keyword.
- #39: Call `create_cm_ticket()` in a `try..commit` block so it can't crash `bin/load_acl`.
- #40: Update dot_gorc.example with `[init_commands]`.
- #43: Bugfix in bin/acl to address incorrect exception reference from when exceptions were cleaned up in release 1.2.
- Simplified basic `Commando` example in `docs/index.rst`.
- Simplified activity output in `Commando` base to/from methods
- Replaced all calls to `time.sleep()` with `reactor.callLater()` within `twister` support of the `command_interval` argument to Twisted state machine constructors.
- Added a way to do SSH version detection within `network`
  - Enhanced `test_tcp_port()` to support optional `check_result` and `expected_result` arguments. If `check_result` is set, the first line of output is retreived from the connection and the starting characters must match `expected_result`.
  - Added a `test_ssh()` function to shortcut to check port 22 for a banner. Defaults to SSHv2.
  - SSH auto-detection in `NetDevices` objects now uses `test_ssh()`.

- Added a new `crypt_md5()` password-hashing function.
- Added proper argument signature to `get_netdevices`.
- Updated misnamed `BadPolicerNameError` to `BadPolicerName`
- More and better documentation improvements, including new documentation for `bin/acl_script`.

6.1.5 1.2

- #23: Commando API overhauled and support added for RANCID
  - RANCID is now officially supported as a source for network device metadata. A new RANCID compatibility module has been added at `rancid`, with support for either single or multiple instance configurations. Multiple instances support can be toggled by setting `RANCID_RECURSE_SUBDIRS` to `True`.
  - The following changes have been made to `netdevices`:
    - `NetDevices` can now import from RANCID
    - A new `Vendor` type has been added to `netdevices` to store canonical vendor names as determined by the new setting `VENDOR_MAP`.
    - When `NetDevice` objects are created, the manufacturer attribute is mapped to a dynamic vendor attribute. This is intended to normalize the way that Trigger identifies vendors internally by a single lower-cased word.
    - All `NetDevice` objects now have a `vendor` attribute with their canonical `Vendor` object attached to it.
    - If the `deviceType` attribute is not set, it is determined automatically based on the `vendor` attribute. The default types for each vendor can be customized using `DEFAULT_TYPES`. If a vendor is not specified witihin `DEFAULT_TYPES`, `FALLBACK_TYPE`. will be used.
    - All logical comparisons that onced used the hard-coded value of the `manufacturer` attribute of a device now instead compare against the `vendor` attribute.
    - You may now tell NetDevices not to fetch acls from AclsDB when instantiate you may also do the same for individual NetDevice objects that you manually create
  - The following changes have been made to `cmds`:
    - The `Commando` class been completely redesigned to reduce boilerplate and simplify creation of new command adapters. This is leveraging the changes to `NetDevice` objects, where the vendor name can be expected to always be normalized to a single, lower-cased word. Defining commands to send to devices is as simple as definiing a `to_{vendor}` method, and parsing return results as simple as `from_{vendor}`.
    - All dynamic method lookups are using the normalized vendor name (e.g. cisco, juniper).
    - Base parse/generate lookup can be disabled explicitly in `Commando` subclasses or as an argument to the constructor.
    - `NetACLInfo` adapted to use Commando 2.0
  - The following changes have been made to Trigger's exception handling
    - All exceptions moved to `exceptions` and given docstrings
    - `trigger.acl.exceptions` has been removed
    - All calls to exceptions updated to new-style exceptions
  - A new -v option has been added to `bin/netdev` to support vendor lookups

- #4: Support for SSH auto-detection and pty/async improvements:
  - The following changes have been made to `twister`:
    - Detection of remotely closed SSH connections so `bin/gong` users can be properly notified (e.g. ssh_exchange_identification errors)
    - New `execute` function to automatically choose the best `execute_` function for a given `NetDevice` object, and is now attached to all `NetDevice` objects
    - `execute_ioslike` now determines whether to use SSH or Telnet automatically
    - All pty connection logic moved out of `bin/gong` into `twister` and is exposed as the `connect` function and also attached to all `NetDevice` objects
    - Interactive sessions may now be optionally logged to a file-like object by passing the log_to argument to the `Interactor` constructor
    - `execute_junoscript` now using `execute_generic_ssh`
    - Command interval added to Junoscript channels for consistency
    - Global `NetDevices` import removed from twister; moved to only occur when a telnet channel is created
  - The following changes have been made to `netdevices`:
    - All `NetDevice` objects now have a `execute` method to perform async interaction
    - The `connect` function is now automatically attached to every `NetDevice` object; to get a pty it's as simple as `dev.connect()`.
    - New helper methods added to `NetDevice` objects:
      - SSH functionality methods: `has_ssh()` (port connection test), `can_ssh_async()` (device supports async), `can_ssh_pty()` (device supports pty)
      - `is_ioslike()` to test if a device is IOS-like as specified by `IOSLIKE_VENDORS`.
      - `is_netscreen` to test if a device is a NetScreen firewall
      - `is_reachable` to test if a device responds to a ping
  - The following changes have been made to `settings`:
    - A mapping of officially supported platforms has been defined at `SUPPORTED_PLATFORMS`
    - `VALID_VENDORS` has been renamed to `SUPPORTED_VENDORS`
    - A mapping of officially supported device types has been defined at `SUPPORTED_TYPES`
    - You may now disable telnet fallback by toggling `TELNET_ENABLED`
    - You may now disable SSH for pty or async by vendor/type using `SSH_PTY_DISABLED` and `SSH_ASYNC_DISABLED` respectively
    - `SSH_TYPES` has been removed as it is no longer needed
  - `Commando` experimentally using the new `NetDevice.execute()` method
  - Two new helper functions added to `cli`: `setup_tty_for_pty` and `update_password_and_reconnect`, which modularize functionality that was in bin/gong that didn't seem to fit anywhere else
- #21: The following changes have been made to support A10 hardware and to enhance handling of SSH channels:
  - Added a new generic SSH channel. The NetScreen and A10 channels are based from this. Further abstraction needed to roll NetScaler channel into this as well.

  - Added a new `execute_generic_ssh` factory function.
  - Refactored `execute_netscreen` to use `execute_generic_ssh`
  - Added a new `execute_ioslike_ssh` factory function utilizing the generic SSH channel to support SSH on IOS-like devices (Brocade, Cisco, Arista, A10, etc.). Works like a charm except for the Brocade VDX.
  - The `Commando` was updated to support A10, NetScreen. Brocade, Arista changed to use SSH vs. telnet.
  - All prompt-matching patterns moved to top of `trigger.twister` as constants
  - A10 added to `IOSLIKE_VENDORS`
- #24: `bin/gong` will now display the reason when it fails to connect to a device.

6.1.6 1.1

- All changes from release 1.0.0.100 (oh hey, duh) are officially part of this release
- #9: Fixed missing imports from `bin/acl_script` and removed a bunch of duplicated code already within the Trigger libs.
- Added new keywords to setup.py
- Some new utilities added to `tools` for merging new access into an existing ACL object
- #17: `RangeList` now sorts port range tuples when parsing access-lists.
- #8: `get_device_password` user-friendly message moved to `pty_connect` so it no longer bleeds into non-interactive usage.
- #15: `output_ios` updated to support optional `acl_name` argument for cases when you need to output a `Term` separately from an `ACL` object. `check_access`, `bin/check_access`, and `bin/find_access` also had to be updated to utilize this new argument.
- #19: `check_access` updated to support 'complicated' checks against Juniper firewall terms with a 'port' statement defined.

6.1.7 1.0.0.100

- `conf` converted from a module to a package.
- All global default settings are now baked into trigger.conf.settings
- `settings` and `autoacl` may now be imported without the proper expected config files in place on disk. If the config files cannot be found, default versions of these objects will be returned.
- All trigger modules can now be imported with default values (but don't try instantiating any objects without following the install instructions!)
- #2: Fixed a bug in `Tacacsrc` where newly-created .tacacsrc files were world-readable. Correct 0600 perms are now enforced on every write().
- #3: Added the ability for :class:~trigger.twister.IoslikeSendExpect' to handle confirmation prompts (such as when a device asks you "are you sure? [y/N]:" by detecting common cases within the prompt-matching logic.
- #5: Added ability for gong –oob to lookup devices by partial hostnames using `device_match()`.
- #6: The `get_firewall_db_conn()` function was moved out of `settings.py` and into `Queue` where it belongs.
- #7: Updated `has_ioslike_error()` to support Brocade VDX errors.

6.1.8 1.0.0.90

- Added support for .gorc file to specify commands to run when using gong to login to a device. Unique commands cand be specified for each vendor.
- Default realm for credentials within .tacacsrc can now be specified within settings.DEFAULT_REALM
- The following changes have been made to trigger.tacacsrc:
  - New module-level update_credentials() function added to facilitate updating of cached user credentials by client applications (e.g. gong)
  - Renamed the exceptions within trigger.tacacsrc to be more human-readable
  - Tacacsrc._parse_old() completely redesigned with real error-handling for bad/missing passwords (GPG-parsing coming "Soon")
  - New Tacacsrc.update_creds() method used to facilitate update of stored credentials within .tacacsrc
  - Realm is now stored as an attribute on Credentials objects to simplify loose-coupling of device/realm information while passing around credentials.
  - prompt_credentials() refactored to be more user-friendly.
  - Blank passwords can no longer be stored within .tacacsrc.
- The following changes have been made to trigger.twister:
  - trigger.twister internals have been updated to support the passing of a list of initial_commands to execute on a device upon logging in.
  - TriggerClientFactory now reads the default realm from settings.DEFAULT_REALM when populating credentials.
  - TriggerClientFactory credentials detection improved
  - All referencing of username/password from credentials by index replaced with attributes.
  - Failed logins via telnet/ssh will now raise a LoginFailure exception that can be handled by client applications (such as gong)
- bin/gong now detects login failures and prompts users to update their cached password.

6.1.9 1.0.0.80

- Typo fix in sample conf/trigger_settings.py
- Explicit imports from trigger.acl and a little docstring cleanup in bin/optimizer
- trigger.acl.autoacl.autoacl() now takes optional explicit_acls as 2nd argument, a set of ACL names, so that we can reference explicit_acls within autoacl() implicit ACL logic, but we don't have to rely on the internals.
- trigger.acl.db.AclsDB.get_acl_set() modified to populate explicit_acls before implicit_acls. autoacl() is now called with these explicit_acls as the 2nd argument.
- Sample autoacl.py in conf/autoacl.py updated to support explicit_acls and a simple example of how it could be used.
- Added support for Juniper "family inet" filters in trigger.acl.parser.
- ACL objects now have a family attribute to support this when constructed or parsed using the .output_junos() method.

6.1.10 1.0.0.70

- Minor bugfix in trigger.netdevices._parse_xml()

6.1.11 1.0.0.60

- New nd2json.py nad nd2sqlite.py tools for use in converting existing netdevices.xml implementations
- Added sample netdevices.json in conf/netdevices.json
- Added SQLite database schema for netdevices in conf/netdevices.sql

6.1.12 1.0.0.50

- New NetDevices device metadata source file support for JSON, XML, or SQLite3
- Companion changes made to conf/trigger_settings.py
- trigger.netdevice.NetDevice objects can now be created on their own and have the minimum set of attributes defaulted to None upon instantiation

6.1.13 1.0.0.40

- Public release!
- Arista and Dell command execution and interactive login support in trigger.twister!

6.1.14 Legacy Versions

Trigger was renumbered to version 1.0 when it was publicly released on April 2, 2012. This legacy version history is incompleted, but is kept here for posterity.

1.6.1

- Users credentials from tacacsrc.Tacacsrc are now stored as a namedtuple aptly named 'Credentials'

1.6.0 - 2011-10-26

- Fixed missing acl.parse import in bin/find_access
- More documentation cleanup!
- The following changes have been made to trigger.cmds.Commando:
  - Added parse/generate methods for Citrix NetScaler devices
  - Renamed Commando.work to Commando.jobs to avoid confusing inside of Commando._add_worker()
  - Added distinct parse/generate methods for each supported vendor type (new: Brocade, Foundry, Citrix)
  - Generate methods are no longer called each time _setup_callback() is called; they are now called once an entry is popped from the jobs queue.
  - All default parse/generate methods now reference base methods to follow DRY in this base class.
- Fixed incorrect IPy.IP import in bin/acl_script

- Trigger.twister.pty_connect will only prompt for distinct passwors on firewalls
- Added _cleanup() method to acl.parser.RangeList objects to allow for addition of lists of mixed lists/tuples/digits and still account for more complex types such as Protocol objects
- Performance tweak to Rangelist._expand() method for calculating ranges.
- Added parsing support for remark statements in IOS numbered ACLs

1.5.9 - 2011-08-17

- Tons and tons of documentation added into the docs folder including usage, API, and setup/install documentation.
- Tons of code docstrings added or clarified across the entire package.
- Added install_requires to setup() in setup.py; removed bdist_hcm install command.
- The following changes have been made to trigger.twister:
  - Massive, massive refactoring.
  - New base class for SSH channels.
  - New NetScaler SSH channel. (Full NetScaler support!)
  - New execute_netscaler() factory function.
  - execute_netscreenlike() renamed to execute_netscreen().
  - Every class method now has a docstring.
  - Many, many things moved around and organized.
- Added doctsrings to trigger.netdevices.NetDevice class methods
- The following CLI scripts have been removed from Trigger packaging to an internal repo & removed from setup.py. (These may be added back after further internal code review.)
  - bin/acl_mass_delete
  - bin/acl_mass_insert
  - bin/fang
  - bin/get_session
  - bin/merge_acls
- The following CLI scripts have had their documentation/attributions updated:
  - bin/fe
  - bin/gong
  - bin/load_acl
- Restructuring within bin/load_acl to properly abstract fetching of on-call engineer data and CM ticket creation into trigger_settings.py.
- External release sanitization:
  - Template for trigger_settings.py updated and internal references removed.
  - Sanitized autoacl.py and added generic usage examples.
- The following items have been moved from bin/load_acl into trigger.utils.cli:

  - NullDevice, print_severed_head, min_sec, pretty_time.
- Fixed a bug in trigger.utils.rcs.RCS that would cause RCS log printing to fail.
- Added REDIS_PORT, REDIS_DB to trigger_settings.py and tweaked trigger.acl.db to support it.
- Fixed bug in bin/netdev causing a false positive against search options.
- trigger.netscreen: Tweak EBNF slightly to parse policies for ScreenOS 6.x.

1.5.8 - 20011-06-08

- trigger.acl.parser fully supports Brocade ACLs now, including the ability to strip comments and properly include the "ip rebind-receive-acl" or "ip rebind-acl" commands.
- trigger.acl.Term objects have a new output_ios_brocade() method to support Brocade-special ACLs
- bin/load_acl will automatically strip comments from Brocade ACLs

1.5.7 - 2011-06-01

- Where possible replaced ElementTree with cElementTree for faster XML parsing
- New NetDevices.match() method allows for case-insensitive queries for devices.
- NetDevices.search() now accepts optional field argument but defaults to nodeName.
- New trigger.acl.ACL.strip_comments() method ... strips... comments... from ACL object.
- bin/fang:
  - Now accepts hostnames as arguments
  - Now *really* properly parses hops on Brocade devices.
- bin/load_acl:
  - Now fully supports Brocade devices.
  - Strips comments from Brocade ACLs prior to staging and load.
  - Now displays temporary log file location to user.
- Removed jobi, orb, nms modules from Trigger; replaced with python-aol versions.

1.5.6 - 2011-05-24

- bin/acl: corrected excpetion catching, changes option help text and made -a and -r append
- bin/gnng, bin/netdev: Added -N flag to toggle production_only flag to NetDevices
- trigger.cmds/trigger.twister: Added support for 'BROCADE' vendor (it's ioslike!)
- trigger.cmds.Commando: All generate_* methods are now passed a device object as the first argument to allow for better dynamic handling of commands to execute
- bin/fang: Can now properly parse hops on Brocade devices.

1.5.5 - 2011-04-27

- bin/acl: Will now tell you when something isn't found
- bin/acl: Added -q flag to silence messages if needed
- get_terminal_width() moved to trigger.utils.cli
- trigger.tacacsrc: Fixed bogus AssertionError for bad .tacacsrc file. Clarified error.
- trigger.twister: Fixed bug in Dell password prompt matching in execute_ioslike()
- bin/fang: Increased default timeout to 30 seconds when collecting devices.
- trigger.cmds.Commando:
  - Replaced all '__foo()' with '_foo()'
  - Removed Commando constructor args that are not used at this time
  - Added production_only flag to Commando constructor

1.5.4 - 2011-03-09

- Fixed a bug in trigger.cmds.Commando that would prevent reactor loop from continuing after an exception was thrown.
- trigger.cmds.Commando now has configurable timeout value (defaults to 30 seconds)
- trigger.acl.tools now looks at acl comments for trigger: make discard
- fixed a bug with gong connecting to devices' oob

1.5.3 - 2011-01-12

- Fixed a bug in trigger.cmds.NetACLInfo where verbosity was not correctly toggled.
- gong (go) will now connect to non-prod devices and throw a warning to the user
- gong can connect to a device through oob by passing the -o or –oob option.
- acl will make any device name lower case before associating an acl with it.

1.5.2 - 2010-11-03

- bin/find_access: Added -D and -S flags to exclude src/dst of 'any' from search results. Useful for when you need to report on inclusive networks but not quite as inclusive as 0.0.0.0/0.
- Fixed a bug in acls.db where a device without an explicit association would return None and throw a ValueError that would halt NetDevices construction.
- Added __hash__() to NetDevice objects so they can be serialized (pickled)
- Fixed a bug in explicit ACL associations that would sometimes return incorrect results
- trigger.cmds.NetACLInfo now has a verbosity toggle (defaults to quiet)
- Caught an exception thrown in NetACLInfo for some Cisco devices

1.5.1 - 2010-09-08

- trigger.conf: import_path() can now be re-used by other modules to load modules from file paths without needing to modify sys.path.
- autoacl can now be loaded from a location specified in settings.AUTOACL_FILE allowing us to keep the ever-changing business rules for acl/device mappings out of the Trigger packaging.
- netdevices:
  - Slight optimization to NetDevice attribute population
  - Added new fields to NetDevice.dump() output
  - All incoming fields from netdevices.xml now normalized
- bin/netdev:
  - added search option for Owning Team (-o)
  - search opt for OnCall Team moved to -O
  - search opt for Owning Org (cost center) moved to -C
  - added search option for Budget Name (-B)
  - refactored search argument parsing code
- bin/fang:
  - will now not display information for ACLs found in settings.IGNORED_ACLS

1.5.0r2 - 2010-08-16

- Minor fix to warnings/shebang for bin/scripts

1.5.0 - 2010-08-04

- acl.db: renamed ExplicitACL to AclsDB, all references adjusted
- process_bulk_loads() moved to trigger.acl.tools
- get_bulk_acls() moved to trigger.acl.tdb
- get_all_acls(), get_netdevices(), populate_bulk_acls() added to trigger.acl.db
- load_acl: now imports bulk_acl functions from trigger.acl.tools
- load_acl: now uses trigger.acl.queue API vs. direct db queries
- load_acl: –bouncy now disables bulk acl thresholding
- load_acl: now displays CM ticket # upon successful completion
- process_bulk_loads() now uses device.bulk_acl associations, better performance
- device_match() now sorts and provides correct choices
- Juniper filter-chain support added to trigger.cmds.NetACLInfo
- gnng updated to use NetACLinfo
- Added proceed() utility function trigger.utils.cli
- Several ACL manipulation functions added to trigger.acl.tools:

  - get_comment_matches() - returns ACL terms comments matching a pattern
  - update_expirations() - updates expiration date for listed ACL terms
  - write_tmpacl() - writes an ACL object to a tempfile
  - diff_files() - returns a diff of two files
  - worklog() - inserts a diff of ACL changes into the ACL worklog
- fang: patched to support Juniper filter-lists

1.4.9r2 - 2010-04-27

- find_access: Corrected missing import for IPy
- tacacsrc.py: Corrected bug with incorrect username association to .tacacsrc in sudo/su use-cases (such as with cron) where login uid differs from current uid.

1.4.9 - 2010-04-26

- You may now use gong (go) to connect to Dell devices (telnet only).
- Completely overhauled tacacsrc.py to support auto-detection of missing .tacacsrc
- Heavily documented all changes to tacacsrc.py
- Twister now imports from tacacsrc for device password fetching
- gen_tacacsrc.py now imports from tacacsrc for .tacacsrc generation
- load_acl now uses get_firewall_db_conn from global settings
- Added new search() method to NetDevices to search on name matches
- Added a new device_match() function to netdevices for use with gong
- gong now uses device_match() to present choices to users
- netdev now uses device_match() to present choices to users

1.4.8 - 2010-04-16

- acls.db replaced with redis key/value store found at trigger.acl.db
- trigger.acl converted to package
- all former trigger.acl functionality under trigger.acl.parser
- autoacls.py moved to trigger.acl.autoacls
- aclscript.py moved to trigger.acl.tools.py
- netdevices.py now using trigger.acl.db instead of flat files
- added trigger.netdevices.NetDevices.all() as shortcut to itervalues()
- You may now use gong (go) to connect to non-TACACS devices, such as OOB or unsupported devices using password authentication.
- The ACL parser has been reorganized slightly to make future modifications more streamlined.
- Load_acl now logs *all* activity to a location specified in Trigger config file.

- Added new 'trigger.utils' package to contain useful modules/operations
- 'acl' command moved into Trigger package
- 'netdev' command moved into Trigger package
- Merged trigger.commandscheduler into trigger.nms
- Basic trigger_settings.py provided in conf directory in source dist.

6.2 Trigger Development

The Trigger developement team is currently a one-man operation led by Jathan McCollum, aka `jathanism`.

6.2.1 Contributing

There are several ways to get involved with Trigger:

- **Use Trigger and send us feedback!** This is the best and easiest to improve the project – let us know how you currently use Trigger and how you want to use it. (Please search the ticket tracker first, though, when submitting feature ideas.)
- **Report bugs.** If you use Trigger and think you've found a bug, check on the ticket tracker to see if anyone's reported it yet, and if not – file a bug! If you can, please try to make sure you can replicate the problem, and provide us with the info we need to reproduce it ourselves (what version of Trigger you're using, what platform you're on, and what exactly you were doing when the bug cropped up.)
- **Submit patches or new features.** Make a Github account, create a fork of the main Trigger repository, and submit a pull request.

All contributors will receive proper attribution for their work. We want to give credit where it is due!

Communication

If an issue ticket exists for a given issue, **please** keep all communication in that ticket's comments. Otherwise, please use whatever avenue of communication works best for you!

Style

Trigger tries very diligently to honor PEP-8, especially (but not limited to!) the following:

- Keep all lines under 80 characters. This goes for the ReST documentation as well as code itself.
  - Exceptions are made for situations where breaking a long string (such as a string being `print`-ed from source code, or an especially long URL link in documentation) would be kind of a pain.
- Typical Python 4-space (soft-tab) indents. No tabs! No 8 space indents! (No 2- or 3-space indents, for that matter!)
- `CamelCase` class names, but `lowercase_underscore_separated` everything else.

6.2.2 Branching/Repository Layout

While Trigger's development methodology isn't set in stone yet, the following items detail how we currently organize the Git repository and expect to perform merges and so forth. This will be chiefly of interest to those who wish to follow a specific Git branch instead of released versions, or to any contributors.

- Completed feature work is merged into the `master` branch, and once enough new features are done, a new release branch is created and optionally used to create prerelease versions for testing – or simply released as-is.
- While we try our best not to commit broken code or change APIs without warning, as with many other open-source projects we can only have a guarantee of stability in the release branches. Only follow `master` (or, even worse, feature branches!) if you're willing to deal with a little pain.
- Bugfixes are to be performed on release branches and then merged into `master` so that `master` is always up-to-date (or nearly so; while it's not mandatory to merge after every bugfix, doing so at least daily is a good idea.)

6.2.3 Releases

We use semantic versioning. Version numbers should follow this format:

```
{Major version}.{Minor version}.{Revision number}.{Build number (optional)}
```

Major

Major releases update the first number, e.g. going from 0.9 to 1.0, and indicate that the software has reached some very large milestone.

For example, the 1.0 release signified a commitment to a medium to long term API and some significant backwards incompatible (compared to the 0.9 series) features. Version 2.0 might indicate a rewrite using a new underlying network technology or an overhaul to be more object-oriented.

Major releases will often be backwards-incompatible with the previous line of development, though this is not a requirement, just a usual happenstance. Users should expect to have to make at least some changes to their `settings.py` when switching between major versions.

Minor

Minor releases, such as moving from 1.0 to 1.1, typically mean that one or more new, large features has been added. They are also sometimes used to mark off the fact that a lot of bug fixes or small feature modifications have occurred since the previous minor release. (And, naturally, some of them will involve both at the same time.)

These releases are guaranteed to be backwards-compatible with all other releases containing the same major version number, so a `settings.py` that works with 1.0 should also work fine with 1.1 or even 1.9.

Bugfix/tertiary

The third and final part of version numbers, such as the '3' in 1.0.3, generally indicate a release containing one or more bugfixes, although minor feature modifications may (rarely) occur.

This third number is sometimes omitted for the first major or minor release in a series, e.g. 1.2 or 2.0, and in these cases it can be considered an implicit zero (e.g. 2.0.0).

To illustrate how Trigger works, here are some basic examples of leveraging the API.

For these examples to work you must have already *installed* and *configured* Trigger, so if you haven't already please do that first!

6.3 Simple Examples

6.3.1 Working with metadata

Get a count of all your devices:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> len(nd)
5539
```

(Whoa! That's a lot!) Let's look up a device.

```
>>> dev = nd.find('edge1-abc')
>>> dev.vendor, dev.deviceType
(<Vendor: Juniper>, 'ROUTER')
>>> dev.has_ssh()
True
```

6.3.2 Get an interactive shell

Since this device has SSH, let's connect to it:

```
>>> dev = nd.find('edge1-abc')
>>> dev.connect()
Connecting to edge1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
--- JUNOS 10.2S6.3 built 2011-01-22 20:06:27 UTC
jathan@edge1-abc>
```

6.3.3 Work with access-lists

Let's start with a simple Cisco ACL:

```
>>> from trigger.acl import parse
>>> aclobj = parse("""access-list 123 permit tcp any host 10.20.30.40 eq 80""")
>>> aclobj.terms
[<Term: None>]
```

And convert it to Juniper format:

```
>>> aclobj.name_terms() # Juniper policy terms must have names
>>> aclobj.terms
[<Term: T1>]
>>> print '\n'.join(aclobj.output(format='junos'))
filter 123 {
    term T1 {
        from {
            destination-address {
                10.20.30.40/32;
```

```
            }
            protocol tcp;
            destination-port 80;
        }
        then {
            accept;
        }
    }
}
```

6.3.4 Cache your login credentials

Trigger will encrypt and store your credentials in a file called `.tacacsrc` in your home directory. We already had them cached in the previous examples, so I removed it and then:

```
>>> from trigger.tacacsrc import Tacacsrc
>>> tcrc = Tacacsrc()
/home/jathan/.tacacsrc not found, generating a new one!

Updating credentials for device/realm 'tacacsrc'
Username: jathan
Password:
Password (again):
>>> tcrc.creds['aol']
Credentials(username='jathan', password='boguspassword', realm='tacacsrc')
```

Passwords can be cached by realm. By default this realm is `'aol'`, but you can change that in the settings. Your credentials are encrypted and decrypted using a shared key. A more secure experimental GPG-encrypted method is in the works.

6.3.5 Login to a device using the `gong` script

Trigger includes a simple tool for end-users to connect to devices called `gong`. (It should be just `go` but we're in the future, so...):

```
$ gong foo1-cisco
Connecting to foo1-cisco.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
foo1-cisco#
foo1-cisco#show clock
20:52:05.777 UTC Sat Jun 23 2012
foo1-cisco#
```

Partial hostnames are supported, too:

```
$ gong foo1
2 possible matches found for 'foo1':
[ 1] foo1-abc.net.aol.com
[ 2] foo1-xyz.net.aol.com
[ 0] Exit

Enter a device number: 2
Connecting to foo1-xyz.net.aol.com.  Use ^X to exit.
```

```
Fetching credentials from /home/jathan/.tacacsrc
foo1-xyz#
```

6.4 Slightly Advanced Examples

6.4.1 Execute commands asynchronously using Twisted

This is a little more advanced... so we saved it for last.

Trigger uses Twisted, which is a callback-based event loop. Wherever possible Twisted's implementation details are abstracted away, but the power is there for those who choose to wield it. Here's a super simplified example of how this might be accomplished:

```
from trigger.netdevices import NetDevices
from twisted.internet import reactor

nd = NetDevices()
dev = nd.find('foo1-abc')

def print_result(data):
    """Display results from a command"""
    print 'Result:', data

def stop_reactor(data):
    """Stop the event loop"""
    print 'Stopping reactor'
    if reactor.running:
        reactor.stop()

# Create an event chain that will execute a given list of commands on this
# device
async = dev.execute(['show clock'])

# When we get results from the commands executed, call this
async.addCallback(print_result)

# Once we're out of commands, or we an encounter an error, call this
async.addBoth(stop_reactor)

# Start the event loop
reactor.run()
```

Which outputs:

```
Result: ['21:27:46.435 UTC Sat Jun 23 2012\n']
Stopping reactor
```

Observe, however, that this only communicated with a single device.

6.4.2 Execute commands asynchronously using the Commando API

`Commando` tries to hide Twisted's implementation details so you don't have to deal with callbacks, while also implementing a worker pool so that you may easily communicate with multiple devices in parallel.

This is a base class that is intended to be extended to perform the operations you desire. Here is a basic example of how we might perform the same example above using `Commando` instead, but also communicating with a second device in parallel:

```
from trigger.cmds import Commando

class ShowClock(Commando):
    """Execute 'show clock' on a list of Cisco devices."""
    vendors = ['cisco']
    commands = ['show clock']

if __name__ == '__main__':
    device_list = ['foo1-abc.net.aol.com', 'foo2-xyz.net.aol.com']
    showclock = ShowClock(devices=device_list)
    showclock.run() # Commando exposes this to start the event loop

    print '\nResults:'
    print showclock.results
```

Which outputs:

```
Sending ['show clock'] to foo2-xyz.net.aol.com
Sending ['show clock'] to foo1-abc.net.aol.com
Received ['21:56:44.701 UTC Sat Jun 23 2012\n'] from foo2-xyz.net.aol.com
Received ['21:56:44.704 UTC Sat Jun 23 2012\n'] from foo1-abc.net.aol.com

Results:
{
    'foo1-abc.net.aol.com': {
        'show clock': '21:56:44.704 UTC Sat Jun 23 2012\n'
    },
    'foo2-xyz.net.aol.com': {
        'show clock': '21:56:44.701 UTC Sat Jun 23 2012\n'
    }
}
```

6.5 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!!

6.5.1 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.

- MySQL-python (MySQLdb)

6.5.2 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

Trigger expects to find its configuration files to be in `/etc/trigger`. This can be customized using the `PREFIX` configuration variable within `settings.py`:

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

That's it! Now you're ready to configure Trigger.

6.5.3 Basic Configuration

For these steps you'll need to download the Trigger tarball, expand it, and then navigate to the root directory (the same directory in which you'll find `setup.py`).

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

Trigger's `autoacl` module expects to find `autoacl.py` in the `PREFIX`. This is used to customize the automatic ACL associations for network devices.

```
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 metadata file

Trigger's `netdevices` module expects to find the device metadata file in `PREFIX`. This is used to customize the automatic ACL associations for network devices.

For the purpose of basic config, we'll just use the sample `netdevices.xml` file:

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

6.5.4 Verifying Functionality

Once the dependencies are installed, fire up your trusty Python interpreter in interactive mode and try doing stuff.

Important: Throughout this documentation you will see commands or code preceded by a triple greater-than prompt (>>>). This indicates that they are being entered into a Python interpreter in *interactive mode*.

To start Python in *interactive mode*, it's as simple as executing `python` from a command prompt:

```
% python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

For more information, please see the official Python documentation on interactive mode.

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'])
```

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

Command-line Tools

Blah blah blah command-line stuff here.

The following tools are included:

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

acl_script - Modify ACLs from the command-line

About **acl_script** is a tool and an API shipped that allows for the quick and easy modifications of filters based on various criteria. This is used most in an automated fashion, allowing for users to quickly and efficiently setup small scripts to auto-generate various portions of an ACL.

Usage Here is the usage output:

```
usage: acl_script [options]

ACL modify/generator from the commandline.
options:
  -h, --help
  -aACL, --acl=ACL      specify the acl file
  -n, --no-changes      don't make the changes
  --show-mods           show modifications being made in a simple format.
  --no-worklog          don't make a worklog entry
  -N, --no-input        require no input (good for scripts)
  -sSOURCE_ADDRESS, --source-address=SOURCE_ADDRESS
                        define a source address
  -dDESTINATION_ADDRESS, --destination-address=DESTINATION_ADDRESS
                        define a destination address
  --destination-address-from-file=DESTINATION_ADDRESS_FROM_FILE
                        read a set of destination-addresses from a file
  --source-address-from-file=SOURCE_ADDRESS_FROM_FILE
                        read a set of source-addresses from a file
  --protocol=PROTOCOL   define a protocol
  -pSOURCE_PORT, --source-port=SOURCE_PORT
                        define a source-port
  --source-port-range=SOURCE_PORT_RANGE
                        define a source-port range
  --destination-port-range=DESTINATION_PORT_RANGE
                        define a destination-port range
  -PDESTINATION_PORT, --destination-port=DESTINATION_PORT
                        define a destination port
  -tMODIFY_SPECIFIC_TERM, --modify-specific-term=MODIFY_SPECIFIC_TERM
```

```
                        When modifying a JUNOS type ACL, you may specify this
                        option one or more times to define a specific JUNOS
                        term you want to modify. This takes one argument which
                        should be the name of term.
  -cMODIFY_BETWEEN_COMMENTS, --modify-between-comments=MODIFY_BETWEEN_COMMENTS
                        When modifying a IOS type ACL, you may specify this
                        option one or more times to define a specific AREA of
                        the ACL you want to modify. You must have at least 2
                        comments defined in the ACL prior to running. This
                        requires two arguments, the start comment, and the end
                        comment. Your modifications will be done between the
                        two.
  --insert-defined      This option works differently based on the type of ACL
                        we are modifying. The one similar characteristic is
                        that this will never remove any access already defined,
                        just append.
  --replace-defined     This option works differently based on the type of ACL
                        we are modifying. The one similar characteristic is
                        that access can be removed, since this replaces whole
                        sets of defined data.
```

Examples

Understanding `--insert-defined` This flag will tell `acl_script` to append (read: never remove) information to a portion of an ACL.

Junos On a Junos-type ACL using `--insert-defined`, this will only replace parts of the term that have been specified on the command-line. This may sound confusing but this example should clear things up.

Take the following term:

```
term sr31337 {
    from {
        source-address {
            10.0.0.0/8;
            11.0.0.0/8;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

If you run `acl_script` with the following arguments:

```
acl_script --modify-specific-term sr31337 --source-address 5.5.5.5/32 --destination-port 81 --insert-
```

The following is generated:

```
term sr31337 {
    from {
        source-address {
            5.5.5.5/32;
            10.0.0.0/8;
            11.0.0.0/8;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80-81;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

As you can see `5.5.5.5/32` was added to the `source-address` portion, and `81` was added as a `destination-port`. Notice that all other fields were left alone.

IOS-like On IOS-like ACLs `--insert-defined` behaves a little bit differently. In this case the `acl_script` will only add access where it is needed.

Take the following example:

```
!!! I AM L33T
access-list 101 permit udp host 192.168.0.1 host 192.168.1.1 eq 80
access-list 101 permit ip host 192.168.0.5 host 192.168.1.10
access-list 101 permit tcp host 192.168.0.6 host 192.168.1.11 eq 22
!!! I AM NOT L33T
```

If you run `acl_script` with the following arguments:

```
acl_script --modify-between-comments "I AM L33T" "I AM NOT L33T" \
  --source-address 192.168.0.5 \
  --destination-address 192.168.1.10 \
  --destination-address 192.168.1.11 \
  --protocol tcp \
  --destination-port 80 \
  --insert-defined
```

This output is generated:

```
!!! I AM L33T
access-list 101 permit udp host 192.168.0.1 host 192.168.1.1 eq 80
access-list 101 permit ip host 192.168.0.5 host 192.168.1.10
access-list 101 permit tcp host 192.168.0.6 host 192.168.1.11 eq 22
access-list 101 permit tcp host 192.168.0.5 host 192.168.1.11 eq 80
!!! I AM NOT L33T
```

As you can see the last line was added, take note that the `192.168.0.5->192.168.1.10:80` access was not added because it was already permitted previously.

Understanding `--replace-defined` This flag will completely replace portions of an ACL with newly-defined information.

Junos Take the following term:

```
term sr31337 {
    from {
        source-address {
            10.0.0.0/8;
            11.0.0.0/8;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

With the following arguments to `acl_script`:: acl_script –modify-specific-term sr31337 –source-address 5.5.5.5 –replace-defined

The following is generated:

```
term sr31337 {
    from {
        source-address {
            5.5.5.5/32;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

IOS-like More on this later!

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™.

check_access - ACL Access Checker

About **check_access** determines if access is already in an ACL and if not provides the output to add.

Usage Here is the usage signature:

```
Usage: check_access [opts] file source dest [protocol [port]]
```

Examples Let's start with a simple Cisco extended ACL called `acl.abc123` that looks like this:

```
% cat acl.abc123
no ip access-list extended abc123
ip access-list extended abc123
!
!!! Permit this network
permit tcp 10.17.18.0 0.0.0.31 any
!
!!! Default deny
deny ip any any
```

Let's use the example flow of checking whether http (port 80/tcp) is permitted from any source to the destination `10.20.30.40` in the policy `acl.abc123`:

```
% check_access acl.abc123 any 10.20.30.40 tcp 80
!
!!! Permit this network
permit tcp 10.17.18.0 0.0.0.31 any
! check_access: ADD THIS TERM
permit tcp any host 10.20.30.40 eq 80
!
!!! Default deny
deny ip any any
```

It adds a comment that says "`check_access: ADD THIS TERM`", followed by the policy one would need to add, and where (above the explicit deny).

Now if it were permitted, say if we chose `10.17.18.19` as the source, it would tell you something different:

```
% check_access acl.acb123 10.17.18.19 10.20.30.40 tcp 80
!
!!! Permit this network
!  check_access: PERMITTED HERE
permit tcp 10.17.18.0 0.0.0.31 any
!
!!! Default deny
deny ip any any
No edits needed.
```

It adds a comment that says "`check_access:  PERMITTED HERE`", followed by the policy that matches the flow. Additionally at the end it also reports "`No edits needed`".

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:
;
;     A10: a10
;  Arista: arista
; Brocade: brocade
;   Cisco: cisco
;  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:
    term mon
    who

arista:
    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>
```

Troubleshooting

Authentication failures If `gong` fails to connect, it tries to tell you why, and in the event of an authentication failure it will give you the opportunity to update your stored credentials:

```
Fetching credentials from /home/j/jathan/.tacacsrc

Connection failed for the following reason:

'\n\n% Authentication failed.\n\n\nUser Access Verification\n\nUsername:'

Authentication failed, would you like to update your password? (Y/n)
```

Blank passwords When initially caching credentials, your password cannot be blank. If you try, `gong` complains:

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

Password cannot be blank, try again!
```

If `gong` detects a blank password in an existing `.tacacsrc` file, it will force you to update it:

```
Missing password for 'aol', initializing...

Updating credentials for device/realm 'aol'
Username [jathan]:
```

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' metdata.

Options:
  --version             show program's version number and exit
  -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).
  -v <VENDOR>, --vendor=<VENDOR>
                        For use with -s:  Match on canonical vendor name.
  -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

        Vendor:            Juniper (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 -v 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
```

Configuration and defaults

This document describes the configuration options available for Trigger.

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.

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 /etc/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([])
```

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")]
```

VENDOR_MAP New in version 1.2. A mapping of manufacturer attribute values to canonical vendor name used by Trigger. These single-word, lowercased canonical names are used throughout Trigger.

If your internal definition differs from the UPPERCASED ones specified below (which they probably do, customize them here.

Default:

```
{
    'A10 NETWORKS': 'a10',
    'ARISTA NETWORKS': 'arista',
    'BROCADE': 'brocade',
    'CISCO SYSTEMS': 'cisco',
    'CITRIX': 'citrix',
    'DELL': 'dell',
    'FOUNDRY': 'foundry',
    'JUNIPER': 'juniper',
    'NETSCREEN TECHNOLOGIES': 'netscreen',
}
```

SUPPORTED_PLATFORMS New in version 1.2. A dictionary keyed by manufacturer name containing a list of the device types for each that is officially supported by Trigger. Do not modify this unless you know what you're doing!

Default:

```
{
    'a10': ['SWITCH'],
    'arista': ['SWITCH'],
    'brocade': ['ROUTER', 'SWITCH'],
    'cisco': ['ROUTER', 'SWITCH'],
    'citrix': ['SWITCH'],
    'dell': ['SWITCH'],
    'foundry': ['ROUTER', 'SWITCH'],
    'juniper': ['FIREWALL', 'ROUTER', 'SWITCH'],
    'netscreen': ['FIREWALL']
}
```

SUPPORTED_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:

```
('a10', 'arista', 'brocade', 'cisco', 'citrix', 'dell', 'foundry',
'juniper', 'netscreen')
```

SUPPORTED_TYPES A tuple of device types officially supported by Trigger. Do not modify this unless you know what you're doing!

Default:

```
('FIREWALL', 'ROUTER', 'SWITCH')
```

DEFAULT_TYPES New in version 1.2. A mapping of of vendor names to the default device type for each in the event that a device object is created and the `deviceType` attribute isn't set for some reason.

Default:

```
{
    'a10': 'SWITCH',
    'arista': 'SWITCH',
    'brocade': 'SWITCH',
    'citrix': 'SWITCH',
    'cisco': 'ROUTER',
    'dell': 'SWITCH',
    'foundry': 'SWITCH',
    'juniper': 'ROUTER',
    'netscreen': 'FIREWALL',
}
```

FALLBACK_TYPE New in version 1.2. When a vendor is not explicitly defined within `DEFAULT_TYPES`, fallback to this type.

Default:

```
'ROUTER'
```

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
```

TELNET_ENABLED New in version 1.2. Whether or not to allow telnet fallback. Set to `False` to disable support for telnet.

Default:

```
True
```

SSH_PTY_DISABLED New in version 1.2. A mapping of vendors to the types of devices for that vendor for which you would like to disable interactive (pty) SSH sessions, such as when using `bin/gong`.

Default:

```
{
    'dell': ['SWITCH'],
}
```

SSH_ASYNC_DISABLED New in version 1.2. A mapping of vendors to the types of devices for that vendor for which you would like to disable asynchronous (NON-interactive) SSH sessions, such as when using `execute` or `Commando` to remotely control a device.

Default:

```
{
    'arista': ['SWITCH'],
    'brocade': ['SWITCH'],
    'dell': ['SWITCH'],
}
```

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:

```
('a10', 'arista', 'brocade', 'cisco', '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 `json`, `rancid`, `sqlite`, `xml`. This MUST match the actual format of `NETDEVICES_FILE` or it won't work for obvious reasons.

Please note that RANCID support is experimental. If you use it you must specify the path to the RANCID directory.

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, RANCID, a SQLite3 database, or XML. You must set `NETDEVICES_FORMAT` to match the type of data.

Please note that RANCID support is experimental. If you use it you must specify the path to the RANCID directory.

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

Default:

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

RANCID_RECURSE_SUBDIRS New in version 1.2. When using RANCID as a data source, toggle whether to treat the RANCID root as a normal instance, or as the root to multiple instances.

You may override this location by setting the `RANCID_RECURSE_SUBDIRS` environment variable to any `True` value.

Default:

```
False
```

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:

```
()
```

JUNIPER_FULL_COMMIT_FIELDS Fields and values defined here will dictate which Juniper devices receive a `commit-configuration full` when populating `commit_commands`. The fields and values must match the objects exactly or it will fallback to `commit-configuration`.

Example:

```
# Perform "commit full" on all Juniper EX4200 switches.
JUNIPER_FULL_COMMIT_FIELDS = {
    'deviceType': 'SWITCH',
    'make': 'EX4200',
}
```

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

Format: `{'filter_name': max_hits}`

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
```

Notification settings

EMAIL_SENDER New in version 1.2.2. The default email sender for email notifications. It's probably a good idea to make this a no-reply address.

Default:

```
'nobody@not.real'
```

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:

```
[]
```

NOTIFICATION_SENDER New in version 1.2.2. The default sender for integrated notifications. This defaults to the fully-qualified domain name (FQDN) for the local host.

Default:

```
socket.gethostname()
```

SUCCESS_RECIPIENTS New in version 1.2.2. Destinations (hostnames, addresses) to notify when things go well.

Default:

```
[]
```

FAILURE_RECIPIENTS New in version 1.2.2. Destinations (hostnames, addresses) to notify when things go not well.

Default:

```
[]
```

NOTIFICATION_HANDLERS New in version 1.2.2. This is a list of fully-qualified import paths for event handler functions. Each path should end with a callable that handles a notification event and returns `True` in the event of a successful notification, or `None`.

To activate a handler, add it to this list. Each handler is represented by a string: the full Python path to the handler's function name.

Handlers are processed in order. Once an event is succesfully handled, all processing stops so that each event is only handled once.

Until this documentation improves, for a good example of how to create a custom handler, review the source code for `email_handler()`.

Default:

```
[
    'trigger.utils.notifications.handlers.email_handler',
]
```

Determine commands to run upon login using `.gorc`

This is used by *go - Device connector* to execute commands upon login to a device. A user may specify a list of commands to execute for each vendor. If the file is not found, or the syntax is bad, no commands will be passed to the device.

By default, only a very limited subset of root commands are allowed to be specified within the `.gorc`. Any root commands not explicitly permitted will be filtered out prior to passing them along to the device.

The only public interface to this module is `get_init_commands`. Given a `.gorc` That looks like this:

```
cisco:
    term mon
    terminal length 0
    show clock
```

This is what is returned:

```
>>> from trigger import gorc
>>> gorc.get_init_commands('cisco')
['term mon', 'terminal length 0', 'show clock']
```

You may also pass a list of commands as the `init_commands` argument to the `connect` function (or a `NetDevice` object's method of the same name) to override anything specified in a user's `.gorc`:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> dev = nd.find('foo1-abc')
>>> dev.connect(init_commands=['show clock', 'exit'])
Connecting to foo1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
foo1-abc#show clock
```

```
22:48:24.445 UTC Sat Jun 23 2012
foo1-abc#exit
>>>
```

For detailed instructions on how to create a `.gorc`, please see *Executing commands upon login*.

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.

Your Source Data Before you can work with device metadata, you must tell Trigger how and from where to read it. You may either modify the values for these options within `settings.py` or you may specify the values as environment variables of the same name as the configuration options.

Please see *Configuration and defaults* for more information on how to do this. There are two configuration options that facilitate this:

:NETDEVICES_FILE: The location of the file containing the metadata. Default: `/etc/trigger/netdevices.xml`

:NETDEVICES_FORMAT: The format of the metadata file. Default: `xml`

When you instantiate `NetDevices` the specified file is read and parsed using the specified format. The currently accepted formats are:

- JSON
- RANCID
- Sqlite
- XML

Except when using RANCID as a data source, the contents of your source data 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!

Importing from RANCID New in version 1.2. Experimental support for using a RANCID repository to populate your metadata is now working. We say it's experimental because it is not yet complete. Currently all it does for you is populates the bare minimum set of fields required for basic functionality.

To learn more please visit the section on working with the *RANCID format*.

Supported Formats

XML XML is the slowest method supported by Trigger, but it is currently the default for legacy reasons. At some point we will switch JSON to the default.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Dummy version of netdevices.xml, with just one real entry modeled 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>
```

Please see `conf/netdevices.xml` within the Trigger source distribution for a full example.

JSON JSON is the fastest method supported by Trigger. This is especially the case if you utilize the optional C extension of `simplejson`. The file can be minified and does not need to be indented.

Here is a sample of what the `netdevices.json` file bundled with the Trigger source code looks like (pretty-printed for readabilty):

```
[
    {
        "adminStatus": "PRODUCTION",
        "enablePW": "boguspassword",
        "OOBTerminalServerTCPPort": "5005",
        "assetID": "0000012345",
        "OOBTerminalServerNodeName": "ts1",
        "onCallEmail": "nobody@aol.net",
        "onCallID": "17",
        "OOBTerminalServerFQDN": "ts1.oob.aol.com",
        "owner": "12345678 - Network Engineering",
        "OOBTerminalServerPort": "5",
        "onCallName": "Data Center",
        "nodeName": "test1-abc.net.aol.com",
        "make": "M40 INTERNET BACKBONE ROUTER",
        "budgetCode": "1234578",
        "budgetName": "Data Center",
        "operationStatus": "MONITORED",
        "deviceType": "ROUTER",
        "lastUpdate": "2010-07-19 19:56:32.0",
        "authMethod": "tacacs",
        "projectName": "Test Lab",
        "barcode": "0101010101",
        "site": "LAB",
        "loginPW": null,
        "lifecycleStatus": "INSTALLED",
        "manufacturer": "JUNIPER",
        "layer3": "1",
        "layer2": "1",
        "room": "CR10",
        "layer4": "1",
        "serialNumber": "987654321",
        "owningTeam": "Data Center",
        "coordinate": "16ZZ",
        "model": "M40-B-AC",
        "OOBTerminalServerConnector": "C"
    },
    ...
]
```

To use JSON, create your `NETDEVICES_FILE` full of objects that look like the one above and set `NETDEVICES_FORMAT` to `'json'`.

Please see `conf/netdevices.json` within the Trigger source distribution for a full example.

RANCID This is the easiest method to get running assuming you've already got a RANCID instance to leverage. At this time, however, the metadata available from RANCID is very limited and populates only the following fields for each `Netdevice` object:

nodeName The device hostname.

manufacturer The representative name of the hardware manufacturer. This is also used to dynamically populate the `vendor` attribute on the device object

vendor The canonical vendor name used internally by Trigger. This will always be a single, lowercased word, and is automatically set when a device object is created.

deviceType One of ('SWITCH', 'ROUTER', 'FIREWALL'). This is currently a hard-coded value for each manufacturer.

adminStatus If RANCID says the device is `'up'`, then this is set to `'PRODUCTION'`; otherwise it's set to `'NON-PRODUCTION'`.

The support for RANCID comes in two forms: single or multiple instance.

Single instance is the default and expects to find the `router.db` file and the `configs` directory in the root directory you specify.

Multiple instance will instead walk the root directory and expect to find `router.db` and `configs` in each subdirectory it finds. Multiple instance can be toggled by seting the value of `RANCID_RECURSE_SUBDIRS` to `True` to your `settings.py`.

To use RANCID as a data source, set the value of `NETDEVICES_FILE` in `settings.py` to the absolute path of location of of the root directory where your RANCID data is stored and set the value `NETDEVICES_FORMAT` to `'rancid'`.

Note: Make sure that the value of `RANCID_RECURSE_SUBDIRS` matches the RANCID method you are using. This setting defaults to `False`, so if you only have a single RANCID instance, there is no need to add it to your `settings.py`.

Lastly, to illustrate what a `NetDevice` object that has been populated by RANCID looks like, here is the output of `.dump()`:

```
Hostname:          test1-abc.net.aol.com
Owning Org.:       None
Owning Team:       None
OnCall Team:       None

Vendor:            Juniper (juniper)
Make:              None
Model:             None
Type:              ROUTER
Location:          None None None

Project:           None
Serial:            None
Asset Tag:         None
Budget Code:       None (None)

Admin Status:      PRODUCTION
Lifecycle Status:  None
Operation Status:  None
Last Updated:      None
```

Compare that to what a device dump looks like when fully populated from CMDB metadata in *What's in a NetDevice?*. It's important to keep this in mind, because if you want to do device associations using any of the unpopulated fields, you're gonna have a bad time. This is subject to change as RANCID support evolves, but this is the way it is for now.

SQLite SQLite is somewhere between JSON and XML as far as performance, but also comes with the added benefit that support is built into Python, and you get a real database file you can leverage in other ways outside of Trigger.

```
--
-- Table structure for table 'netdevices'
--
-- This is for 'netdevices.sql' SQLite support within
-- trigger.netdevices.NetDevices for storing and tracking network device
-- metadata.
--
-- This is based on the current set of existing attributes in use and is by no
-- means exclusive. Feel free to add your own fields to suit your environment.
--

CREATE TABLE netdevices (
    id  INTEGER PRIMARY KEY,
    OOBTerminalServerConnector VARCHAR(1024),
    OOBTerminalServerFQDN VARCHAR(1024),
    OOBTerminalServerNodeName VARCHAR(1024),
    OOBTerminalServerPort VARCHAR(1024),
    OOBTerminalServerTCPPort VARCHAR(1024),
    acls VARCHAR(1024),
    adminStatus VARCHAR(1024),
    assetID VARCHAR(1024),
    authMethod VARCHAR(1024),
    barcode VARCHAR(1024),
    budgetCode VARCHAR(1024),
    budgetName VARCHAR(1024),
    bulk_acls VARCHAR(1024),
    connectProtocol VARCHAR(1024),
    coordinate VARCHAR(1024),
    deviceType VARCHAR(1024),
    enablePW VARCHAR(1024),
    explicit_acls VARCHAR(1024),
    gslb_master VARCHAR(1024),
    implicit_acls VARCHAR(1024),
    lastUpdate VARCHAR(1024),
    layer2 VARCHAR(1024),
    layer3 VARCHAR(1024),
    layer4 VARCHAR(1024),
    lifecycleStatus VARCHAR(1024),
    loginPW VARCHAR(1024),
    make VARCHAR(1024),
    manufacturer VARCHAR(1024),
    model VARCHAR(1024),
    nodeName VARCHAR(1024),
    onCallEmail VARCHAR(1024),
    onCallID VARCHAR(1024),
    onCallName VARCHAR(1024),
    operationStatus VARCHAR(1024),
    owner VARCHAR(1024),
    owningTeam VARCHAR(1024),
    projectID VARCHAR(1024),
    projectName VARCHAR(1024),
    room VARCHAR(1024),
    serialNumber VARCHAR(1024),
    site VARCHAR(1024)
);
```

To use SQLite, create a database using the schema provided within Trigger source distribution at `conf/netdevices.sql`. You will need to populate the database full of rows with the columns above and set

NETDEVICES_FORMAT to 'sqlite'.

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

1. The NETDEVICES_FILE file must be readable and must properly parse using the format specified by NETDEVICES_FORMAT (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 there can only be one instance of this object 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 each time one had to parse NETDEVICES_FILE all over again.

Upon startup, each device object/element/row found within NETDEVICES_FILE 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 _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 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 metadata files 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

        Vendor:            Juniper (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.vendor, dev.deviceType
('test1-abc.net.aol.com', <Vendor: 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(vendor='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>
```

Managing Credentials with .tacacsrc

About The `tacacsrc` module provides an abstract interface to the management and storage of user credentials in the `.tacacsrc` file. This is used throughout Trigger to automatically retrieve credentials for a user whenever they connect to devices.

How it works The `Tacacsrc` class is the core interface for encrypting credentials when they are stored, and decrypting the credentials when they are retrieved. A unique `.tacacsrc` file is stored in each user's home directory, and is forcefully set to be readable only (permissions: `0400`) by the owning user.

There are two implementations, the first of which is the only one that is officially supported at this time, and which is properly documented.

1. Shared key encryption

   This method is the default. It relies on a shared key to be stored in a file somewhere on the system. The location of this file can be customized in `settings.py` using `TACACSRC_KEYFILE`.

   This method has a glaring security flaw in that anyone who discerns the location of the keyfile can see the passphrase used for the encryption. This risk is mitigated somewhat by ensuring that each user's `.tacacsrc` has strict file permissions.

2. GPG encryption

   This method is experimental but is intended to be the long-term replacement for the shared key method. To enable GPG encryption, set `USE_GPG_AUTH` to `True` within `settings.py`.

   This method is very secure because there is no centralized passphrase used for encryption. Each user chooses their own.

Usage

Creating a .tacacsrc When you create an instance of `Tacacsrc`, it will try to read the `.tacacsrc` file. If this file is not found, or cannot be properly parsed, it will be initialized:

```
>>> from trigger import tacacsrc
>>> tcrc = tacacsrc.Tacacsrc()
/home/jathan/.tacacsrc not found, generating a new one!

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

If you inspect the `.tacacsrc` file, you'll see that both the username and password are encrypted:

```
% cat ~/.tacacsrc
# Saved by trigger.tacacsrc at 2012-06-23 11:38:51 PDT

aol_uname_ = uiXq7eHEq2A=
aol_pwd_ = GUpzkuFJfN8=
```

Retrieving stored credentials Credentials can be cached by realm. By default this realm is `'aol'`, but you can change that in `settings.py` using `DEFAULT_REALM`. Credentials are stored as a dictionary under the `.creds` attribute, keyed by the realm for each set of credentials:

```
>>> tcrc.creds
{'aol': Credentials(username='jathan', password='boguspassword', realm='aol')}
```

There is also a module-level function, `get_device_password()`, that takes a realm name as an argument, which will instantiate `Tacacsrc` for you and returns the credentials for the realm:

```
>>> tacacsrc.get_device_password('aol')
Credentials(username='jathan', password='boguspassword', realm='aol')
```

Updating stored credentials The module-level function `update_credentials()` will prompt a user to update their stored credentials. It expects the realm key you would like to update and an optional username you would like to use. If you don't specify a user, the existing username for the realm is kept.

```
>>> tacacsrc.update_credentials('aol')

Updating credentials for device/realm 'aol'
Username [jathan]:
Password:
Password (again):

Credentials updated for user: 'jathan', device/realm: 'aol'.
True
>>> tcrc.creds
{'aol': Credentials(username='jathan', password='panda', realm='aol')}
```

This function will return `True` upon a successful update to `.tacacsrc`.

Using GPG encryption **EXPERIMENTAL! PROCEED AT YOUR OWN RISK!! FEEDBACK WELCOME!!**

Before you proceed, you must make sure to have gpg2 and gpg-agent installed on your system.

Enabling GPG In `settings.py` set `USE_GPG_AUTH` to `False`.

Generating your GPG key Execute:

```
gpg2 --gen-key
```

When asked fill these in with the values appropriate for you:

```
Real name: jathan
Email address: jathan.mccollum@teamaol.com
Comment: Jathan McCollum
```

It will confirm:

```
You selected this USER-ID:
    "jathan (Jathan McCollum) <jathan@marduk.itsec.aol.com>"
```

Here is a snippet to try and make this part of the core API, but is not yet implemented:

```
>>> import os, pwd, socket
>>> pwd.getpwnam(os.getlogin()).pw_gecos
'Jathan McCollum'
>>> socket.gethostname()
'wtfpwn.bogus.aol.com'
>>> h = socket.gethostname()
>>> u = os.getlogin()
>>> n = pwd.getpwnam(u).pw_gecos
>>> e = '%s@%s' % (u,h)
>>> print '%s (%s) <%s>' % (u,n,e)
jathan (Jathan McCollum) <jathan@wtfpwn.bogus.aol.com'
```

Convert your tacacsrc to GPG Assuming you already have a "legacy" `.tacacsrc` file, execute:

```
tacacsrc2gpg.py
```

It will want to generate your GPG key. This can take a VERY LONG time. We need a workaround for this.

And then it outputs:

```
This will overwrite your .tacacsrc.gpg and all gnupg configuration, are you sure? (y/N)
Would you like to convert your OLD tacacsrc configuration file to your new one? (y/N)
Converting old tacacsrc to new kind :)
OLD
/opt/bcs/packages/python-modules-2.0/lib/python/site-packages/simian/tacacsrc.py:125: DeprecationWarn
  (fin,fout) = os.popen2('gpg2 --yes --quiet -r %s -e -o %s' % (self.username, self.file_name))
```

Update your gpg.conf Trigger should also do this for us, but alas...

Add `'use-agent'` to `~/.gnupg/gpg.conf`:

```
echo 'use-agent\n' > .gnupg/gpg.conf
```

This will allow you to unlock your GPG store at the beginning of the day, and have the gpg-agent broker the communication encryption/decryption of the file for 24 hours.

See if it works

1. Connect to a device.
2. It will prompt for passphrase
3. ...and connected! (aka Profit)

Other utilities You may check if a user has a GPG-enabled credential store:

```
>>> from trigger import tacacsrc
>>> tcrc = tacacsrc.Tacacsrc()
>>> tcrc.user_has_gpg()
False
```

Convert `.tacacsrc` to `.tacacsrc.gpg`:

```
>>> tacacsrc.convert_tacacsrc()
```

6.5.5 Integrated Load Queue

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 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
```

Important: Throughout this documentation you will see commands or code preceded by a triple greater-than prompt (>>>). This indicates that they are being entered into a Python interpreter in *interactive mode*.

To start Python in *interactive mode*, it's as simple as executing `python` from a command prompt:

```
% python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

For more information, please see the official Python documentation on interactive mode.

6.6 License

6.7 Overview

This document is a high-level overview of Trigger's features and a little history about why it exists in the first place.

6.7.1 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 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.

6.7.2 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 AOL 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 caused 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.

6.7.3 History

Trigger was 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 configuration management operations, so we decided to ditch the acronym and just go with a name that more accurately hints at what it does.

6.7.4 Components

Trigger is comprised of the following core components:

NetDevices

An abstract interface to network device metadata and security policy associations.

Twister

Asynchronous device interaction library. Performs login and basic command-line interaction support via telnet or SSH using the Twisted asynchronous I/O framework.

Access-List Parser

An ACL parsing library which 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.

- Converting ACLs from one format to another (e.g. Cisco to Juniper)
- Testing an ACL to determine is access is permitted
- Automatically associate ACLs to devices by metatdata

Change Management

An abstract interface to bounce windows and moratoria. Includes support for RCS version-control system for maintaining configuration data and an integrated automated task queue.

Commands

Command execution library which 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.

TACACSrc

Network credentials library that provides an abstract interface to storing user credentials encrypted on disk.

Command-Line Tools

Trigger includes a suite of tools for simplifying many common tasks, including:

- Quickly get an interactive shell
- Simple metadata search tool

6.8 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!

6.8.1 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

acl_script - Modify ACLs from the command-line

About

acl_script is a tool and an API shipped that allows for the quick and easy modifications of filters based on various criteria. This is used most in an automated fashion, allowing for users to quickly and efficiently setup small scripts to auto-generate various portions of an ACL.

Usage

Here is the usage output:

```
usage: acl_script [options]

ACL modify/generator from the commandline.
options:
  -h, --help
  -aACL, --acl=ACL      specify the acl file
  -n, --no-changes      don't make the changes
  --show-mods           show modifications being made in a simple format.
  --no-worklog          don't make a worklog entry
  -N, --no-input        require no input (good for scripts)
  -sSOURCE_ADDRESS, --source-address=SOURCE_ADDRESS
                        define a source address
```

```
  -dDESTINATION_ADDRESS, --destination-address=DESTINATION_ADDRESS
                        define a destination address
  --destination-address-from-file=DESTINATION_ADDRESS_FROM_FILE
                        read a set of destination-addresses from a file
  --source-address-from-file=SOURCE_ADDRESS_FROM_FILE
                        read a set of source-addresses from a file
  --protocol=PROTOCOL   define a protocol
  -pSOURCE_PORT, --source-port=SOURCE_PORT
                        define a source-port
  --source-port-range=SOURCE_PORT_RANGE
                        define a source-port range
  --destination-port-range=DESTINATION_PORT_RANGE
                        define a destination-port range
  -PDESTINATION_PORT, --destination-port=DESTINATION_PORT
                        define a destination port
  -tMODIFY_SPECIFIC_TERM, --modify-specific-term=MODIFY_SPECIFIC_TERM
                        When modifying a JUNOS type ACL, you may specify this
                        option one or more times to define a specific JUNOS
                        term you want to modify. This takes one argument which
                        should be the name of term.
  -cMODIFY_BETWEEN_COMMENTS, --modify-between-comments=MODIFY_BETWEEN_COMMENTS
                        When modifying a IOS type ACL, you may specify this
                        option one or more times to define a specific AREA of
                        the ACL you want to modify. You must have at least 2
                        comments defined in the ACL prior to running. This
                        requires two arguments, the start comment, and the end
                        comment. Your modifications will be done between the
                        two.
  --insert-defined      This option works differently based on the type of ACL
                        we are modifying. The one similar characteristic is
                        that this will never remove any access already defined,
                        just append.
  --replace-defined     This option works differently based on the type of ACL
                        we are modifying. The one similar characteristic is
                        that access can be removed, since this replaces whole
                        sets of defined data.
```

Examples

Understanding `--insert-defined` This flag will tell `acl_script` to append (read: never remove) information to a portion of an ACL.

Junos On a Junos-type ACL using `--insert-defined`, this will only replace parts of the term that have been specified on the command-line. This may sound confusing but this example should clear things up.

Take the following term:

```
term sr31337 {
    from {
        source-address {
            10.0.0.0/8;
            11.0.0.0/8;
        }
        destination-address {
            192.168.0.1/32;
        }
```

```
        destination-port 80;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

If you run `acl_script` with the following arguments:

```
acl_script --modify-specific-term sr31337 --source-address 5.5.5.5/32 --destination-port 81 --insert-
```

The following is generated:

```
term sr31337 {
    from {
        source-address {
            5.5.5.5/32;
            10.0.0.0/8;
            11.0.0.0/8;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80-81;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

As you can see `5.5.5.5/32` was added to the `source-address` portion, and `81` was added as a `destination-port`. Notice that all other fields were left alone.

IOS-like On IOS-like ACLs `--insert-defined` behaves a little bit differently. In this case the `acl_script` will only add access where it is needed.

Take the following example:

```
!!! I AM L33T
access-list 101 permit udp host 192.168.0.1 host 192.168.1.1 eq 80
access-list 101 permit ip host 192.168.0.5 host 192.168.1.10
access-list 101 permit tcp host 192.168.0.6 host 192.168.1.11 eq 22
!!! I AM NOT L33T
```

If you run `acl_script` with the following arguments:

```
acl_script --modify-between-comments "I AM L33T" "I AM NOT L33T" \
  --source-address 192.168.0.5 \
  --destination-address 192.168.1.10 \
  --destination-address 192.168.1.11 \
  --protocol tcp \
  --destination-port 80 \
  --insert-defined
```

This output is generated:

```
!!! I AM L33T
access-list 101 permit udp host 192.168.0.1 host 192.168.1.1 eq 80
access-list 101 permit ip host 192.168.0.5 host 192.168.1.10
access-list 101 permit tcp host 192.168.0.6 host 192.168.1.11 eq 22
access-list 101 permit tcp host 192.168.0.5 host 192.168.1.11 eq 80
!!! I AM NOT L33T
```

As you can see the last line was added, take note that the `192.168.0.5->192.168.1.10:80` access was not added because it was already permitted previously.

Understanding `--replace-defined` This flag will completely replace portions of an ACL with newly-defined information.

Junos Take the following term:

```
term sr31337 {
    from {
        source-address {
            10.0.0.0/8;
            11.0.0.0/8;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

With the following arguments to `acl_script`: acl_script –modify-specific-term sr31337 –source-address 5.5.5.5 –replace-defined

The following is generated:

```
term sr31337 {
    from {
        source-address {
            5.5.5.5/32;
        }
        destination-address {
            192.168.0.1/32;
        }
        destination-port 80;
        protocol tcp;
    }
    then {
        accept;
        count sr31337;
    }
}
```

IOS-like More on this later!

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™.

check_access - ACL Access Checker

About

check_access determines if access is already in an ACL and if not provides the output to add.

Usage

Here is the usage signature:

```
Usage: check_access [opts] file source dest [protocol [port]]
```

Examples

Let's start with a simple Cisco extended ACL called `acl.abc123` that looks like this:

```
% cat acl.abc123
no ip access-list extended abc123
ip access-list extended abc123
!
!!! Permit this network
permit tcp 10.17.18.0 0.0.0.31 any
!
!!! Default deny
deny ip any any
```

Let's use the example flow of checking whether http (port 80/tcp) is permitted from any source to the destination `10.20.30.40` in the policy `acl.abc123`:

```
% check_access acl.abc123 any 10.20.30.40 tcp 80
!
!!! Permit this network
permit tcp 10.17.18.0 0.0.0.31 any
! check_access: ADD THIS TERM
permit tcp any host 10.20.30.40 eq 80
!
!!! Default deny
deny ip any any
```

It adds a comment that says "`check_access: ADD THIS TERM`", followed by the policy one would need to add, and where (above the explicit deny).

Now if it were permitted, say if we chose `10.17.18.19` as the source, it would tell you something different:

```
% check_access acl.acb123 10.17.18.19 10.20.30.40 tcp 80
!
!!! Permit this network
!  check_access: PERMITTED HERE
permit tcp 10.17.18.0 0.0.0.31 any
!
!!! Default deny
deny ip any any
No edits needed.
```

It adds a comment that says "`check_access: PERMITTED HERE`", followed by the policy that matches the flow. Additionally at the end it also reports "`No edits needed`".

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:
;
```

```
;     A10: a10
;  Arista: arista
; Brocade: brocade
;   Cisco: cisco
;  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:
    term mon
    who

arista:
    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>
```

Troubleshooting

Authentication failures If `gong` fails to connect, it tries to tell you why, and in the event of an authentication failure it will give you the opportunity to update your stored credentials:

```
Fetching credentials from /home/j/jathan/.tacacsrc

Connection failed for the following reason:

'\n\n% Authentication failed.\n\n\nUser Access Verification\n\nUsername:'

Authentication failed, would you like to update your password? (Y/n)
```

Blank passwords When initially caching credentials, your password cannot be blank. If you try, `gong` complains:

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

Password cannot be blank, try again!
```

If `gong` detects a blank password in an existing `.tacacsrc` file, it will force you to update it:

```
Missing password for 'aol', initializing...

Updating credentials for device/realm 'aol'
Username [jathan]:
```

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' metdata.

Options:
  --version             show program's version number and exit
  -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).
  -v <VENDOR>, --vendor=<VENDOR>
                        For use with -s:  Match on canonical vendor name.
  -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

        Vendor:            Juniper (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 -v 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
```

6.8.2 Configuration and defaults

This document describes the configuration options available for Trigger.

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.

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 /etc/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([])
```

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")]
```

VENDOR_MAP New in version 1.2. A mapping of manufacturer attribute values to canonical vendor name used by Trigger. These single-word, lowercased canonical names are used throughout Trigger.

If your internal definition differs from the UPPERCASED ones specified below (which they probably do, customize them here.

Default:

```
{
    'A10 NETWORKS': 'a10',
    'ARISTA NETWORKS': 'arista',
    'BROCADE': 'brocade',
    'CISCO SYSTEMS': 'cisco',
    'CITRIX': 'citrix',
    'DELL': 'dell',
    'FOUNDRY': 'foundry',
    'JUNIPER': 'juniper',
    'NETSCREEN TECHNOLOGIES': 'netscreen',
}
```

SUPPORTED_PLATFORMS New in version 1.2. A dictionary keyed by manufacturer name containing a list of the device types for each that is officially supported by Trigger. Do not modify this unless you know what you're doing!

Default:

```
{
    'a10': ['SWITCH'],
    'arista': ['SWITCH'],
    'brocade': ['ROUTER', 'SWITCH'],
    'cisco': ['ROUTER', 'SWITCH'],
    'citrix': ['SWITCH'],
    'dell': ['SWITCH'],
    'foundry': ['ROUTER', 'SWITCH'],
    'juniper': ['FIREWALL', 'ROUTER', 'SWITCH'],
    'netscreen': ['FIREWALL']
}
```

SUPPORTED_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:

```
('a10', 'arista', 'brocade', 'cisco', 'citrix', 'dell', 'foundry',
'juniper', 'netscreen')
```

SUPPORTED_TYPES A tuple of device types officially supported by Trigger. Do not modify this unless you know what you're doing!

Default:

```
('FIREWALL', 'ROUTER', 'SWITCH')
```

DEFAULT_TYPES New in version 1.2. A mapping of of vendor names to the default device type for each in the event that a device object is created and the `deviceType` attribute isn't set for some reason.

Default:

```
{
    'a10': 'SWITCH',
    'arista': 'SWITCH',
    'brocade': 'SWITCH',
    'citrix': 'SWITCH',
    'cisco': 'ROUTER',
    'dell': 'SWITCH',
    'foundry': 'SWITCH',
    'juniper': 'ROUTER',
    'netscreen': 'FIREWALL',
}
```

FALLBACK_TYPE New in version 1.2. When a vendor is not explicitly defined within `DEFAULT_TYPES`, fallback to this type.

Default:

```
'ROUTER'
```

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
```

TELNET_ENABLED New in version 1.2. Whether or not to allow telnet fallback. Set to `False` to disable support for telnet.

Default:

```
True
```

SSH_PTY_DISABLED New in version 1.2. A mapping of vendors to the types of devices for that vendor for which you would like to disable interactive (pty) SSH sessions, such as when using `bin/gong`.

Default:

```
{
    'dell': ['SWITCH'],
}
```

SSH_ASYNC_DISABLED New in version 1.2. A mapping of vendors to the types of devices for that vendor for which you would like to disable asynchronous (NON-interactive) SSH sessions, such as when using `execute` or `Commando` to remotely control a device.

Default:

```
{
    'arista': ['SWITCH'],
    'brocade': ['SWITCH'],
    'dell': ['SWITCH'],
}
```

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:

```
('a10', 'arista', 'brocade', 'cisco', '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 `json`, `rancid`, `sqlite`, `xml`. This MUST match the actual format of `NETDEVICES_FILE` or it won't work for obvious reasons.

Please note that RANCID support is experimental. If you use it you must specify the path to the RANCID directory.

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, RANCID, a SQLite3 database, or XML. You must set `NETDEVICES_FORMAT` to match the type of data.

Please note that RANCID support is experimental. If you use it you must specify the path to the RANCID directory.

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

Default:

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

RANCID_RECURSE_SUBDIRS New in version 1.2. When using RANCID as a data source, toggle whether to treat the RANCID root as a normal instance, or as the root to multiple instances.

You may override this location by setting the `RANCID_RECURSE_SUBDIRS` environment variable to any `True` value.

Default:

```
False
```

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:

```
()
```

JUNIPER_FULL_COMMIT_FIELDS Fields and values defined here will dictate which Juniper devices receive a `commit-configuration full` when populating `commit_commands`. The fields and values must match the objects exactly or it will fallback to `commit-configuration`.

Example:

```
# Perform "commit full" on all Juniper EX4200 switches.
JUNIPER_FULL_COMMIT_FIELDS = {
    'deviceType': 'SWITCH',
    'make': 'EX4200',
}
```

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

Format: `{'filter_name': max_hits}`

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
```

Notification settings

EMAIL_SENDER New in version 1.2.2. The default email sender for email notifications. It's probably a good idea to make this a no-reply address.

Default:

```
'nobody@not.real'
```

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:

```
[]
```

NOTIFICATION_SENDER New in version 1.2.2. The default sender for integrated notifications. This defaults to the fully-qualified domain name (FQDN) for the local host.

Default:

```
socket.gethostname()
```

SUCCESS_RECIPIENTS New in version 1.2.2. Destinations (hostnames, addresses) to notify when things go well.

Default:

```
[]
```

FAILURE_RECIPIENTS New in version 1.2.2. Destinations (hostnames, addresses) to notify when things go not well.

Default:

```
[]
```

NOTIFICATION_HANDLERS New in version 1.2.2. This is a list of fully-qualified import paths for event handler functions. Each path should end with a callable that handles a notification event and returns `True` in the event of a successful notification, or `None`.

To activate a handler, add it to this list. Each handler is represented by a string: the full Python path to the handler's function name.

Handlers are processed in order. Once an event is succesfully handled, all processing stops so that each event is only handled once.

Until this documentation improves, for a good example of how to create a custom handler, review the source code for `email_handler()`.

Default:

```
[
    'trigger.utils.notifications.handlers.email_handler',
]
```

6.8.3 Determine commands to run upon login using `.gorc`

This is used by *go - Device connector* to execute commands upon login to a device. A user may specify a list of commands to execute for each vendor. If the file is not found, or the syntax is bad, no commands will be passed to the device.

By default, only a very limited subset of root commands are allowed to be specified within the `.gorc`. Any root commands not explicitly permitted will be filtered out prior to passing them along to the device.

The only public interface to this module is `get_init_commands`. Given a `.gorc` That looks like this:

```
cisco:
    term mon
    terminal length 0
    show clock
```

This is what is returned:

```
>>> from trigger import gorc
>>> gorc.get_init_commands('cisco')
['term mon', 'terminal length 0', 'show clock']
```

You may also pass a list of commands as the `init_commands` argument to the `connect` function (or a `NetDevice` object's method of the same name) to override anything specified in a user's `.gorc`:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> dev = nd.find('foo1-abc')
>>> dev.connect(init_commands=['show clock', 'exit'])
Connecting to foo1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
foo1-abc#show clock
22:48:24.445 UTC Sat Jun 23 2012
foo1-abc#exit
>>>
```

For detailed instructions on how to create a `.gorc`, please see *Executing commands upon login*.

6.8.4 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.

Your Source Data

Before you can work with device metadata, you must tell Trigger how and from where to read it. You may either modify the values for these options within `settings.py` or you may specify the values as environment variables of the same name as the configuration options.

Please see *Configuration and defaults* for more information on how to do this. There are two configuration options that facilitate this:

:NETDEVICES_FILE: The location of the file containing the metadata. Default: `/etc/trigger/netdevices.xml`

:NETDEVICES_FORMAT: The format of the metadata file. Default: `xml`

When you instantiate `NetDevices` the specified file is read and parsed using the specified format. The currently accepted formats are:

- JSON
- RANCID
- Sqlite
- XML

Except when using RANCID as a data source, the contents of your source data 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!

Importing from RANCID

New in version 1.2. Experimental support for using a RANCID repository to populate your metadata is now working. We say it's experimental because it is not yet complete. Currently all it does for you is populates the bare minimum

set of fields required for basic functionality.

To learn more please visit the section on working with the *RANCID format*.

Supported Formats

XML

XML is the slowest method supported by Trigger, but it is currently the default for legacy reasons. At some point we will switch JSON to the default.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Dummy version of netdevices.xml, with just one real entry modeled 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>
```

Please see `conf/netdevices.xml` within the Trigger source distribution for a full example.

JSON

JSON is the fastest method supported by Trigger. This is especially the case if you utilize the optional C extension of `simplejson`. The file can be minified and does not need to be indented.

Here is a sample of what the `netdevices.json` file bundled with the Trigger source code looks like (pretty-printed for readabilty):

```
[
    {
        "adminStatus": "PRODUCTION",
        "enablePW": "boguspassword",
        "OOBTerminalServerTCPPort": "5005",
        "assetID": "0000012345",
        "OOBTerminalServerNodeName": "ts1",
        "onCallEmail": "nobody@aol.net",
        "onCallID": "17",
        "OOBTerminalServerFQDN": "ts1.oob.aol.com",
        "owner": "12345678 - Network Engineering",
        "OOBTerminalServerPort": "5",
        "onCallName": "Data Center",
        "nodeName": "test1-abc.net.aol.com",
        "make": "M40 INTERNET BACKBONE ROUTER",
        "budgetCode": "1234578",
        "budgetName": "Data Center",
        "operationStatus": "MONITORED",
        "deviceType": "ROUTER",
        "lastUpdate": "2010-07-19 19:56:32.0",
        "authMethod": "tacacs",
        "projectName": "Test Lab",
        "barcode": "0101010101",
        "site": "LAB",
        "loginPW": null,
        "lifecycleStatus": "INSTALLED",
        "manufacturer": "JUNIPER",
        "layer3": "1",
        "layer2": "1",
        "room": "CR10",
        "layer4": "1",
        "serialNumber": "987654321",
        "owningTeam": "Data Center",
        "coordinate": "16ZZ",
        "model": "M40-B-AC",
        "OOBTerminalServerConnector": "C"
    },
    ...
]
```

To use JSON, create your `NETDEVICES_FILE` full of objects that look like the one above and set `NETDEVICES_FORMAT` to `'json'`.

Please see `conf/netdevices.json` within the Trigger source distribution for a full example.

RANCID

This is the easiest method to get running assuming you've already got a RANCID instance to leverage. At this time, however, the metadata available from RANCID is very limited and populates only the following fields for each `Netdevice` object:

nodeName The device hostname.

manufacturer The representative name of the hardware manufacturer. This is also used to dynamically populate the `vendor` attribute on the device object

vendor The canonical vendor name used internally by Trigger. This will always be a single, lowercased word, and is automatically set when a device object is created.

deviceType One of ('SWITCH', 'ROUTER', 'FIREWALL'). This is currently a hard-coded value for each manufacturer.

adminStatus If RANCID says the device is `'up'`, then this is set to `'PRODUCTION'`; otherwise it's set to `'NON-PRODUCTION'`.

The support for RANCID comes in two forms: single or multiple instance.

Single instance is the default and expects to find the `router.db` file and the `configs` directory in the root directory you specify.

Multiple instance will instead walk the root directory and expect to find `router.db` and `configs` in each subdirectory it finds. Multiple instance can be toggled by seting the value of `RANCID_RECURSE_SUBDIRS` to `True` to your `settings.py`.

To use RANCID as a data source, set the value of `NETDEVICES_FILE` in `settings.py` to the absolute path of location of of the root directory where your RANCID data is stored and set the value `NETDEVICES_FORMAT` to `'rancid'`.

Note: Make sure that the value of `RANCID_RECURSE_SUBDIRS` matches the RANCID method you are using. This setting defaults to `False`, so if you only have a single RANCID instance, there is no need to add it to your `settings.py`.

Lastly, to illustrate what a `NetDevice` object that has been populated by RANCID looks like, here is the output of `.dump()`:

```
Hostname:          test1-abc.net.aol.com
Owning Org.:       None
Owning Team:       None
OnCall Team:       None

Vendor:            Juniper (juniper)
Make:              None
Model:             None
Type:              ROUTER
Location:          None None None

Project:           None
Serial:            None
Asset Tag:         None
Budget Code:       None (None)

Admin Status:      PRODUCTION
Lifecycle Status:  None
Operation Status:  None
Last Updated:      None
```

Compare that to what a device dump looks like when fully populated from CMDB metadata in *What's in a NetDevice?*. It's important to keep this in mind, because if you want to do device associations using any of the unpopulated fields, you're gonna have a bad time. This is subject to change as RANCID support evolves, but this is the way it is for now.

SQLite

SQLite is somewhere between JSON and XML as far as performance, but also comes with the added benefit that support is built into Python, and you get a real database file you can leverage in other ways outside of Trigger.

```
--
-- Table structure for table 'netdevices'
--
-- This is for 'netdevices.sql' SQLite support within
-- trigger.netdevices.NetDevices for storing and tracking network device
-- metadata.
--
-- This is based on the current set of existing attributes in use and is by no
-- means exclusive. Feel free to add your own fields to suit your environment.
--

CREATE TABLE netdevices (
    id  INTEGER PRIMARY KEY,
    OOBTerminalServerConnector VARCHAR(1024),
    OOBTerminalServerFQDN VARCHAR(1024),
    OOBTerminalServerNodeName VARCHAR(1024),
    OOBTerminalServerPort VARCHAR(1024),
    OOBTerminalServerTCPPort VARCHAR(1024),
    acls VARCHAR(1024),
    adminStatus VARCHAR(1024),
    assetID VARCHAR(1024),
    authMethod VARCHAR(1024),
    barcode VARCHAR(1024),
    budgetCode VARCHAR(1024),
    budgetName VARCHAR(1024),
    bulk_acls VARCHAR(1024),
    connectProtocol VARCHAR(1024),
    coordinate VARCHAR(1024),
    deviceType VARCHAR(1024),
    enablePW VARCHAR(1024),
    explicit_acls VARCHAR(1024),
    gslb_master VARCHAR(1024),
    implicit_acls VARCHAR(1024),
    lastUpdate VARCHAR(1024),
    layer2 VARCHAR(1024),
    layer3 VARCHAR(1024),
    layer4 VARCHAR(1024),
    lifecycleStatus VARCHAR(1024),
    loginPW VARCHAR(1024),
    make VARCHAR(1024),
    manufacturer VARCHAR(1024),
    model VARCHAR(1024),
    nodeName VARCHAR(1024),
    onCallEmail VARCHAR(1024),
    onCallID VARCHAR(1024),
    onCallName VARCHAR(1024),
    operationStatus VARCHAR(1024),
    owner VARCHAR(1024),
    owningTeam VARCHAR(1024),
    projectID VARCHAR(1024),
    projectName VARCHAR(1024),
    room VARCHAR(1024),
    serialNumber VARCHAR(1024),
```

```
    site VARCHAR(1024)
);
```

To use SQLite, create a database using the schema provided within Trigger source distribution at `conf/netdevices.sql`. You will need to populate the database full of rows with the columns above and set `NETDEVICES_FORMAT` to `'sqlite'`.

Getting Started

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

1. The `NETDEVICES_FILE` file must be readable and must properly parse using the format specified by `NETDEVICES_FORMAT` (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 there can only be one instance of this object 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 each time one had to parse `NETDEVICES_FILE` all over again.

Upon startup, each device object/element/row found within `NETDEVICES_FILE` 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 `_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 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 metadata files 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

        Vendor:            Juniper (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.vendor, dev.deviceType
('test1-abc.net.aol.com', <Vendor: 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(vendor='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>
```

6.8.5 Managing Credentials with .tacacsrc

About

The `tacacsrc` module provides an abstract interface to the management and storage of user credentials in the `.tacacsrc` file. This is used throughout Trigger to automatically retrieve credentials for a user whenever they connect to devices.

How it works

The `Tacacsrc` class is the core interface for encrypting credentials when they are stored, and decrypting the credentials when they are retrieved. A unique `.tacacsrc` file is stored in each user's home directory, and is forcefully set to be readable only (permissions: `0400`) by the owning user.

There are two implementations, the first of which is the only one that is officially supported at this time, and which is properly documented.

1. Shared key encryption

   This method is the default. It relies on a shared key to be stored in a file somewhere on the system. The location of this file can be customized in `settings.py` using `TACACSRC_KEYFILE`.

   This method has a glaring security flaw in that anyone who discerns the location of the keyfile can see the passphrase used for the encryption. This risk is mitigated somewhat by ensuring that each user's `.tacacsrc` has strict file permissions.

2. GPG encryption

   This method is experimental but is intended to be the long-term replacement for the shared key method. To enable GPG encryption, set `USE_GPG_AUTH` to `True` within `settings.py`.

   This method is very secure because there is no centralized passphrase used for encryption. Each user chooses their own.

Usage

Creating a .tacacsrc

When you create an instance of `Tacacsrc`, it will try to read the `.tacacsrc` file. If this file is not found, or cannot be properly parsed, it will be initialized:

```
>>> from trigger import tacacsrc
>>> tcrc = tacacsrc.Tacacsrc()
/home/jathan/.tacacsrc not found, generating a new one!

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

If you inspect the `.tacacsrc` file, you'll see that both the username and password are encrypted:

```
% cat ~/.tacacsrc
# Saved by trigger.tacacsrc at 2012-06-23 11:38:51 PDT

aol_uname_ = uiXq7eHEq2A=
aol_pwd_ = GUpzkuFJfN8=
```

Retrieving stored credentials

Credentials can be cached by realm. By default this realm is `'aol'`, but you can change that in `settings.py` using `DEFAULT_REALM`. Credentials are stored as a dictionary under the `.creds` attribute, keyed by the realm for each set of credentials:

```
>>> tcrc.creds
{'aol': Credentials(username='jathan', password='boguspassword', realm='aol')}
```

There is also a module-level function, `get_device_password()`, that takes a realm name as an argument, which will instantiate `Tacacsrc` for you and returns the credentials for the realm:

```
>>> tacacsrc.get_device_password('aol')
Credentials(username='jathan', password='boguspassword', realm='aol')
```

Updating stored credentials

The module-level function `update_credentials()` will prompt a user to update their stored credentials. It expects the realm key you would like to update and an optional username you would like to use. If you don't specify a user, the existing username for the realm is kept.

```
>>> tacacsrc.update_credentials('aol')

Updating credentials for device/realm 'aol'
Username [jathan]:
Password:
Password (again):

Credentials updated for user: 'jathan', device/realm: 'aol'.
True
>>> tcrc.creds
{'aol': Credentials(username='jathan', password='panda', realm='aol')}
```

This function will return `True` upon a successful update to `.tacacsrc`.

Using GPG encryption

EXPERIMENTAL! PROCEED AT YOUR OWN RISK!! FEEDBACK WELCOME!!

Before you proceed, you must make sure to have gpg2 and gpg-agent installed on your system.

Enabling GPG

In `settings.py` set `USE_GPG_AUTH` to `False`.

Generating your GPG key

Execute:

```
gpg2 --gen-key
```

When asked fill these in with the values appropriate for you:

```
Real name: jathan
Email address: jathan.mccollum@teamaol.com
Comment: Jathan McCollum
```

It will confirm:

```
You selected this USER-ID:
    "jathan (Jathan McCollum) <jathan@marduk.itsec.aol.com>"
```

Here is a snippet to try and make this part of the core API, but is not yet implemented:

```
>>> import os, pwd, socket
>>> pwd.getpwnam(os.getlogin()).pw_gecos
'Jathan McCollum'
>>> socket.gethostname()
'wtfpwn.bogus.aol.com'
>>> h = socket.gethostname()
>>> u = os.getlogin()
>>> n = pwd.getpwnam(u).pw_gecos
>>> e = '%s@%s' % (u,h)
>>> print '%s (%s) <%s>' % (u,n,e)
jathan (Jathan McCollum) <jathan@wtfpwn.bogus.aol.com'
```

Convert your tacacsrc to GPG

Assuming you already have a "legacy" `.tacacsrc` file, execute:

```
tacacsrc2gpg.py
```

It will want to generate your GPG key. This can take a VERY LONG time. We need a workaround for this.

And then it outputs:

```
This will overwrite your .tacacsrc.gpg and all gnupg configuration, are you sure? (y/N)
Would you like to convert your OLD tacacsrc configuration file to your new one? (y/N)
Converting old tacacsrc to new kind :)
OLD
/opt/bcs/packages/python-modules-2.0/lib/python/site-packages/simian/tacacsrc.py:125: DeprecationWarn
  (fin,fout) = os.popen2('gpg2 --yes --quiet -r %s -e -o %s' % (self.username, self.file_name))
```

Update your gpg.conf

Trigger should also do this for us, but alas...

Add `'use-agent'` to `~/.gnupg/gpg.conf`:

```
echo 'use-agent\n' > .gnupg/gpg.conf
```

This will allow you to unlock your GPG store at the beginning of the day, and have the gpg-agent broker the communication encryption/decryption of the file for 24 hours.

See if it works

1. Connect to a device.
2. It will prompt for passphrase
3. ...and connected! (aka Profit)

Other utilities

You may check if a user has a GPG-enabled credential store:

```
>>> from trigger import tacacsrc
>>> tcrc = tacacsrc.Tacacsrc()
>>> tcrc.user_has_gpg()
False
```

Convert `.tacacsrc` to `.tacacsrc.gpg`:

```
>>> tacacsrc.convert_tacacsrc()
```

6.9 API Documentation

Trigger's core API is made up of several components. For a more detailed explanation of these components, please see the *Overview*.

6.9.1 `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.vendor
<Vendor: 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=None*, *acl_name=None*)
Output term to IOS traditional format.

Parameters

- **prefix** – Prefix to use, default: 'access-list'
- **acl_name** – Name of access-list to display

output_ios_named(*prefix=''*, **args*, ***kwargs*)
Output term to IOS named format.

output_iosxr(*prefix=''*, **args*, ***kwargs*)
Output term to IOS XR format.

output_junos(**args*, ***kwargs*)
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.

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'*, *acl_name=None*)

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.

Parameters

- **terms_to_check** – A list of Term objects to check
- **new_term** – The Term object used for the access test
- **quiet** – Toggle whether output is displayed
- **format** – The ACL format to use for output display
- **acl_name** – The ACL name to use for output display

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

trigger.acl.tools.**insert_term_into_acl**(*new_term*, *aclobj*, *debug=False*)

Return a new ACL object with the new_term added in the proper place based on the aclobj. Intended to recursively append to an interim ACL object based on a list of Term objects.

It's safe to assume that this function is incomplete pending better documentation and examples.

Parameters

- **new_term** – The Term object to use for comparison against aclobj
- **aclobj** – The original ACL object to use for creation of new_acl

Example:

```
import copy
# terms_to_be_added is a list of Term objects that is to be added in
# the "right place" into new_acl based on the contents of aclobj
original_acl = parse(open('acl.original'))
new_acl = copy.deepcopy(original_acl) # Dupe the original
for term in terms_to_be_added:
    new_acl = generate_new_acl(term, new_acl)
```

trigger.acl.tools.**create_new_acl**(*old_file*, *terms_to_be_added*)
Given a list of Term objects call insert_term_into_acl() to determine what needs to be added in based on the contents of old_file. Returns a new ACL object.

6.9.2 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.

6.9.3 trigger.cmds — Command execution library

This module abstracts the asynchronous execution of commands on multiple network devices. It allows for integrated parsing and event-handling of return data for rapid integration to existing or newly-created tools.

The Commando class is designed to be extended but can still be used as-is to execute commands and return the results as-is.

Please see the source code for ShowClock class for a basic example of one might create a subclass. Better documentation is in the works!

class trigger.cmds.**Commando**(*devices=None*, *commands=None*, *incremental=None*, *max_conns=10*, *verbose=False*, *timeout=30*, *production_only=True*, *allow_fallback=True*)
Execute commands asynchronously on multiple network devices.

This class is designed to be extended but can still be used as-is to execute commands and return the results as-is.

At the bare minimum you must specify a list of `devices` to interact with. You may optionally specify a list of `commands` to execute on those devices, but doing so will execute the same commands on every device regardless of platform.

If `commands` are not specified, they will be expected to be emitted by the `generate` method for a given platform. Otherwise no commands will be executed.

If you wish to customize the commands executed by device, you must define a `to_{vendor_name}` method containing your custom logic.

If you wish to customize what is done with command results returned from a device, you must define a `from_{vendor_name}` method containing your custom logic.

Parameters

- **devices** – A list of device hostnames or `NetDevice` objects
- **commands** – (Optional) A list of commands to execute on the `devices`.
- **incremental** – (Optional) A callback that will be called with an empty sequence upon connection and then called every time a result comes back from the device, with the list of all results.
- **max_conns** – (Optional) The maximum number of simultaneous connections to keep open.
- **verbose** – (Optional) Whether or not to display informational messages to the console.
- **timeout** – (Optional) Time in seconds to wait for each command executed to return a result
- **production_only** – (Optional) If set, includes all devices instead of excluding any devices where `adminStatus` is not set to `PRODUCTION`.
- **allow_fallback** – If set (default), allow fallback to base parse/generate methods when they are not customized in a subclass, otherwise an exception is raised when a method is called that has not been explicitly defined.

errback(*failure*, *device*)

The default errback. Overload for custom behavior but make sure it always decrements the connections.

Parameters

- **failure** – Usually a Twisted `Failure` instance.
- **device** – A `NetDevice` object

generate(*device*, *commands=None*, *extra=None*)

Generate commands to be run on a device. If you don't provide `commands` to the class constructor, this will return an empty list.

Define a 'to_{vendor_name}' method to customize the behavior for each platform.

Parameters

- **device** – A `NetDevice` object
- **commands** – (Optional) A list of commands to execute on the device. If not specified in they will be inherited from commands passed to the class constructor.
- **extra** – (Optional) A dictionary of extra data to send to the generate method for the device.

map_results(*commands=None*, *results=None*)

Return a dict of `{command: result, ...}`

parse(*results*, *device*)
Parse output from a device.

Define a 'from_{vendor_name}' method to customize the behavior for each platform.

Parameters

- **results** – The results of the commands executed on the device
- **device** – A `NetDevice` object

reactor_running
Return whether reactor event loop is running or not

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

select_next_device(*jobs=None*)
Select another device for the active queue.

Currently only returns the next device in the job queue. This is abstracted out so that this behavior may be customized, such as for future support for incremental callbacks.

Parameters **jobs** – (Optional) The jobs queue. If not set, uses `self.jobs`.

Returns A `NetDevice` object

store_error(*device*, *error*)
A simple method for storing an error called by all default parse/generate methods.

If you want to customize the default method for storing results, overload this in your subclass.

Parameters

- **device** – A `NetDevice` object
- **error** – The error to store. Anything you want really, but usually a Twisted `Failure` instance.

store_results(*device*, *results*)
A simple method for storing results called by all default parse/generate methods.

If you want to customize the default method for storing results, overload this in your subclass.

Parameters

- **device** – A `NetDevice` object
- **results** – The results to store. Anything you want really.

to_juniper(*device*, *commands=None*, *extra=None*)
This just creates a series of `<command>foo</command>` elements to pass along to execute_junoscript()

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').

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

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

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

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

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

ipv4_cidr_to_netmask (*bits*)
Convert CIDR bits to netmask

to_arista (*dev*, *commands=None*, *extra=None*)
Similar to IOS, but:

- Arista has no "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

to_brocade(*dev*, *commands=None*, *extra=None*)
This is the "show me all interface information" command we pass to IOS devices

to_cisco(*dev*, *commands=None*, *extra=None*)
This is the "show me all interface information" command we pass to IOS devices

to_foundry(*dev*, *commands=None*, *extra=None*)
This is the "show me all interface information" command we pass to IOS devices

to_juniper(*dev*, *commands=None*, *extra=None*)
Generates an etree.Element object suitable for use with JunoScript

class trigger.cmds.**ShowClock**(*devices=None*, *commands=None*, *incremental=None*, *max_conns=10*, *verbose=False*, *timeout=30*, *production_only=True*, *allow_fallback=True*)
A simple example that runs `show clock` and parses it to `datetime.datetime` object.

from_brocade(*results*, *device*)
Parse Brocade time. Brocade switches and routers behave differently...

from_cisco(*results*, *device*)
Parse Cisco time

6.9.4 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.

class trigger.conf.**BaseSettings**
Common logic for settings whether set by a module or by the user.

6.9.5 trigger.exceptions — Trigger's Exceptions

All custom exceptions used by Trigger. Where possible built-in exceptions are used, but sometimes we need more descriptive errors.

exception trigger.exceptions.**ACLError**
A base exception for all ACL-related errors.

exception trigger.exceptions.**ACLNameError**
A base exception for all ACL naming errors.

exception trigger.exceptions.**ACLSetError**
A base exception for all ACL Set errors.

exception trigger.exceptions.**ActionError**
A base exception for all Term action errors.

exception trigger.exceptions.**BadACLName**
Raised when an ACL object is assigned an invalid name.

exception trigger.exceptions.**BadCounterName**
Raised when a counter name is invalid.

exception trigger.exceptions.**BadForwardingClassName**
Raised when a forwarding-class name is invalid.

exception trigger.exceptions.**BadIPSecSAName**
Raised when an IPSec SA name is invalid.

exception trigger.exceptions.**BadMatchArgRange**
Raised when a match condition argument does not fall within a specified range.

exception trigger.exceptions.**BadPolicerName**
Raised when a policer name is invalid.

exception trigger.exceptions.**BadRejectCode**
Raised when an invalid rejection code is specified.

exception trigger.exceptions.**BadRoutingInstanceName**
Raised when a routing-instance name specified in an action is invalid.

exception trigger.exceptions.**BadTermName**
Raised when an invalid name is assigned to a Term object

exception trigger.exceptions.**BadVendorName**
Raised when a Vendor object has a problem with the name.

exception trigger.exceptions.**CommandFailure**
Raised when a command fails to execute, such as when it results in an error.

exception trigger.exceptions.**CommandTimeout**
Raised when a command times out while executing.

exception trigger.exceptions.**CommandoError**
A base exception for all Commando-related errors.

exception trigger.exceptions.**ConnectionFailure**
Raised when a connection attempt totally fails.

exception trigger.exceptions.**ImproperlyConfigured**
Raised when something is improperly... configured...

exception trigger.exceptions.**InvalidACLSet**
Raised when an invalid ACL set is specified.

exception trigger.exceptions.**IoslikeCommandFailure**
Raised when a command fails on an IOS-like device.

exception trigger.exceptions.**JunoscriptCommandFailure**(*tag*)
Raised when a Junoscript command fails on a Juniper device.

exception trigger.exceptions.**LoginFailure**
Raised when authentication to a remote system fails.

exception trigger.exceptions.**LoginTimeout**
Raised when login to a remote systems times out.

exception trigger.exceptions.**MatchError**
A base exception for all errors related to Term Matches objects.

exception trigger.exceptions.**MissingACLName**
Raised when an ACL object is missing a name.

exception trigger.exceptions.**MissingPlatform**
Raised when a specific device platform is not supported.

exception trigger.exceptions.**MissingTermName**
Raised when a an un-named Term is output to a format that requires Terms to be named (e.g. Juniper).

exception trigger.exceptions.**NetDeviceError**
A base exception for all NetDevices related errors.

exception trigger.exceptions.**NetScreenError**
A general exception for NetScreen devices.

exception trigger.exceptions.**NetScreenParseError**
Raised when a NetScreen policy cannot be parsed.

exception trigger.exceptions.**NetscalerCommandFailure**
Raised when a command fails on a NetScaler device.

exception trigger.exceptions.**NotificationFailure**
Raised when a notification fails and has not been silenced.

exception trigger.exceptions.**ParseError**(*reason*, *line=None*, *column=None*)
Raised when there is an error parsing/normalizing an ACL that tries to tell you where it failed.

exception trigger.exceptions.**SSHConnectionLost**(*code*, *desc*)
Raised when an SSH connection is lost for any reason.

exception trigger.exceptions.**TriggerError**
A base exception for all Trigger-related errors.

exception trigger.exceptions.**TwisterError**
A base exception for all errors related to Twister.

exception trigger.exceptions.**UnknownActionName**
Raised when an action assigned to a ~trigger.acl.parser.Term‘ object is unknown.

exception trigger.exceptions.**UnknownMatchArg**
Raised when an unknown match argument is specified.

exception trigger.exceptions.**UnknownMatchType**
Raised when an unknown match condition is specified.

exception trigger.exceptions.**UnsupportedVendor**
Raised when a vendor is not supported by Trigger.

exception trigger.exceptions.**VendorSupportLacking**
Raised when a feature is not supported by a given vendor.

6.9.6 trigger.gorc — Determine commands to run upon login

This is used by *go - Device connector* to execute commands upon login to a device. A user may specify a list of commands to execute for each vendor. If the file is not found, or the syntax is bad, no commands will be passed to the device.

By default, only a very limited subset of root commands are allowed to be specified within the `.gorc`. Any root commands not explicitly permitted will be filtered out prior to passing them along to the device.

The only public interface to this module is `get_init_commands`. Given a `.gorc` That looks like this:

```
cisco:
    term mon
    terminal length 0
    show clock
```

This is what is returned:

```
>>> from trigger import gorc
>>> gorc.get_init_commands('cisco')
['term mon', 'terminal length 0', 'show clock']
```

You may also pass a list of commands as the `init_commands` argument to the `connect` function (or a `NetDevice` object's method of the same name) to override anything specified in a user's `.gorc`:

```
>>> from trigger.netdevices import NetDevices
>>> nd = NetDevices()
>>> dev = nd.find('foo1-abc')
>>> dev.connect(init_commands=['show clock', 'exit'])
Connecting to foo1-abc.net.aol.com.  Use ^X to exit.

Fetching credentials from /home/jathan/.tacacsrc
foo1-abc#show clock
22:48:24.445 UTC Sat Jun 23 2012
foo1-abc#exit
>>>
```

`trigger.gorc.`**`filter_commands`**(*cmds*)

Filters out root commands that are not explicitly allowed by `ALLOWED_COMMANDS` and returns the filtered list.

Parameters **cmds** – A list of commands that should be filtered

Returns filtered list of commands

`trigger.gorc.`**`get_init_commands`**(*vendor*)

Return a list of init commands for a given vendor name. In all failure cases it will return an empty list.

Parameters **vendor** – A vendor name (e.g. 'juniper')

Returns list of commands

`trigger.gorc.`**`parse_commands`**(*vendor*, *section='init_commands'*, *config=None*)

Fetch the init commands.

Parameters

- **vendors** – A vendor name (e.g. 'juniper')
- **section** – The section of the config
- **config** – A parsed ConfigParser object

Returns list of commands

`trigger.gorc.`**`read_config`**(*filepath='/home/docs/.gorc'*)

Read the .gorc file

Parameters **filepath** – The path to the .gorc file

Returns A parsed ConfigParser object

6.9.7 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.vendor, dev.make
(<Vendor: 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*, *with_acls=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.

can_ssh_async()

Am I enabled to use SSH async?

can_ssh_pty()

Am I enabled to use SSH pty?

dump()

Prints details for a device.

has_ssh()

Am I even listening on SSH?

is_firewall()
Am I a firewall?

is_ioslike()
Am I an IOS-like device (as determined by `settings.IOSLIKE_VENDORS`)?

is_netscaler()
Am I a NetScaler?

is_netscreen()
Am I a NetScreen running ScreenOS?

is_reachable()
Do I respond to a ping?

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.`**Vendor**(*manufacturer=None*)
Map a manufacturer name to Trigger's canonical name.

Given a manufacturer name like 'CISCO SYSTEMS', this will attempt to map it to the canonical vendor name specified in `settings.VENDOR_MAP`. If this can't be done, attempt to split the name up ('CISCO, 'SYSTEMS') and see if any of the words map. An exception is raised as a last resort.

This exposes a normalized name that can be used in the event of a multi-word canonical name.

determine_vendor(*manufacturer*)
Try to turn the provided vendor name into the cname.

normalized
Return the normalized name for the vendor.

class `trigger.netdevices.`**NetDevices**(*production_only=True*, *with_acls=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*, *with_acls*)
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']

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(**myargs)
```

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

6.9.8 `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.`**`NSRawPolicy`**(*data*, *isglobal=0*)
Container for policy definitions.

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

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

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

`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.

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

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

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

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.**NSPolicy**(*name=None*, *address_book=<trigger.netscreen.NSAddressBook object at 0x286fed0>*, *service_book=<trigger.netscreen.NSServiceBook object at 0x3a88250>*, *address_groups=None*, *service_groups=None*, *source_zone='Untrust'*, *destination_zone='Trust'*, *id=0*, *action='permit'*, *isglobal=False*)

Container for individual policy definitions.

6.9.9 trigger.rancid — RANCID Compatibility Library

Parse RANCID db files so they can be converted into Trigger NetDevice objects. New in version 1.2. Far from complete. Very early in development. Here is a basic example.

```
>>> from trigger import rancid
>>> rancid_root = '/path/to/rancid/data'
>>> r = Rancid(rancid_root)
>>> dev = r.devices.get('test1-abc.net.aol.com')
>>> dev
RancidDevice(nodeName='test-abc.net.aol.com', manufacturer='juniper', deviceStatus='up', deviceType=N
```

Another option if you want to get the parsed RANCID data directly without having to create an object is as simple as this:

```
>>> parsed = rancid.parse_rancid_data('/path/to/dancid/data')
```

Or using multiple RANCID instances within a single root:

```
>>> multi_parsed = rancid.parse_rancid_data('/path/to/rancid/data', recurse_subdirs=True)
```

trigger.rancid.**parse_rancid_file**(*rancid_root*, *filename='router.db'*, *fields=None*)

Parse a RANCID file and return generator representing a list of lists mapped to the `fields`.

Parameters

- **rancid_root** – Where to find the file
- **filename** – Name of the file to parse (e.g. `router.db`)
- **fields** – (Optional) A list of field names used to map to the device data

trigger.rancid.**parse_devices**(*metadata*, *parser*)

Iterate device `metadata` to use `parser` to create and return a list of network device objects.

Parameters

- **metadata** – A collection of key/value pairs (Generally returned from `parse_rancid_file`)
- **parser** – A callabale used to create your objects

trigger.rancid.**walk_rancid_subdirs**(*rancid_root*, *config_dirname='configs'*, *fields=None*)

Walk the `rancid_root` and parse the included RANCID files.

Returns a dictionary keyed by the name of the subdirs with values set to the parsed data for each RANCID file found inside.

```
>>> from trigger import rancid
>>> subdirs = rancid.walk_rancid_subdirs('/data/rancid')
>>> subdirs.get('network-security')
{'router.db': <generator object <genexpr> at 0xa5b852c>,
 'routers.all': <generator object <genexpr> at 0xa5a348c>,
```

```
'routers.down': <generator object <genexpr> at 0xa5be9dc>,
'routers.up': <generator object <genexpr> at 0xa5bea54>}
```

Parameters

- **rancid_root** – Where to find the file
- **config_dirname** – If the 'configs' dir is named something else
- **fields** – (Optional) A list of field names used to map to the device data

trigger.rancid.**parse_rancid_data**(*rancid_root*, *filename='router.db'*, *fields=None*, *config_dirname='configs'*, *recurse_subdirs=False*)

Parse single or multiple RANCID instances and return an iterator of the device metadata.

A single instance expects to find 'router.db' in `rancid_root`.

If you set `recurise_subdirs`, multiple instances will be expected, and a `router.db` will be expected to be found in each subdirectory.

Parameters

- **rancid_root** – Where to find the file
- **filename** – Name of the file to parse (e.g. `router.db`)
- **fields** – (Optional) A list of field names used to map to the device data
- **config_dirname** – If the 'configs' dir is named something else
- **recurse_subdirs** – Whether to recurse directories (e.g. multiple instances)

trigger.rancid.**gather_devices**(*subdir_data*, *rancid_db_file='router.db'*)

Returns a chained iterator of parsed RANCID data, based from the results of `walk_rancid_subdirs`.

This iterator is suitable for consumption by `parse_devices` or Trigger's `NetDevices`.

Parameters

- **rancid_root** – Where to find your RANCID files (router.db, et al.)
- **rancid_db_file** – If it's named other than `router.db`

class trigger.rancid.**Rancid**(*rancid_root*, *rancid_db_file='router.db'*, *config_dirname='configs'*, *device_fields=None*, *device_class=None*, *recurse_subdirs=False*)

Holds RANCID data. INCOMPLETE.

Defaults to a single RANID instance specified as `rancid_root`. It will parse the file found at `rancid_db_file` and use this to populate the `devices` dictionary with instances of `device_class`.

If you set `recurse_subdirs`, it is assumed that `rancid_root` holds one or more individual RANCID instances and will attempt to walk them, parse them, and then aggregate all of the resulting device instances into the `devices` dictionary.

Still needs:

•Config parsing for metadata (make, model, type, serial, etc.)

•Recursive Config file population/parsing when `recurse_subdirs` is set

Parameters

- **rancid_root** – Where to find your RANCID files (router.db, et al.)
- **rancid_db_file** – If it's named other than `router.db`

- **config_dir** – If it's named other than `configs`
- **device_fields** – A list of field names used to map to the device data. These must match the attributes expected by `device_class`.
- **device_class** – If you want something other than `RancidDevice`
- **recurse_subdirs** – Whether you want to recurse directories.

class `trigger.rancid.`**`RancidDevice`**
A simple subclass of namedtuple to store contents of parsed RANCID files.

Designed to support all router.* files. The field names are intended to be compatible with Trigger's NetDevice objects.

Parameters

- **nodeName** – Hostname of device
- **manufacturer** – Vendor/manufacturer name of device
- **deviceStatus** – (Optional) Up/down status of device
- **deviceType** – (Optional) The device type... determined somehow

6.9.10 `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).

6.9.11 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**(*log_to=None*)
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 as a state machine.

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.

The method self._setup_channelOpen() should be called by channelOpen() in the subclasses. Before you subclass, however, see if you can't just use TriggerSSHGenericChannel as-is!

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

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

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

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*, *prompt_pattern=None*, *device=None*)
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.**TriggerSSHGenericChannel**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
An SSH channel using all of the Trigger defaults to interact with network devices that implement SSH without any tricks.

Currently A10, Cisco, Brocade, NetScreen can simply use this. Nice!

Before you create your own subclass, see if you can't use me as-is!

class trigger.twister.**TriggerSSHJunoscriptChannel**(*localWindow=0*, *localMaxPacket=0*, *remoteWindow=0*, *remoteMaxPacket=0*, *conn=None*, *data=None*, *avatar=None*)
An SSH channel to execute Junoscript commands on a Juniper device running Junos.

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*)
An SSH channel to interact with Citrix NetScaler hardware.

It's almost a generic SSH channel except that we must check for errors first, because a prompt is not returned when an error is received. This had to be accounted for in the `dataReceived()` method.

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**
SSH transport with Trigger's defaults.

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

connectionLost(*reason*)
Detect when the transport connection is lost, such as when the remote end closes the connection prematurely (hosts.allow, etc.)

Parameters **reason** – A Failure instance containing the error object

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 login state machine. 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.**connect**(*device*, *init_commands=None*, *output_logger=None*, *login_errback=None*, *reconnect_handler=None*)
Connect to a network device via pty for an interactive shell.

Parameters

- **device** – A `NetDevice` object.

- **init_commands** – (Optional) A list of commands to execute upon logging into the device. If not set, they will be attempted to be read from `.gorc`.
- **output_logger** – (Optional) If set all data received by the device, including user input, will be written to this logger. This logger must behave like a file-like object and a implement a `write()` method. Hint: Use `StringIO`.
- **login_errback** – (Optional) An callable to be used as an errback that will handle the login failure behavior. If not set the default handler will be used.
- **reconnect_handler** – (Optional) A callable to handle the behavior of an authentication failure after a login has failed. If not set default handler will be used.

`trigger.twister.`**`execute`**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*)

Connect to a `device` and sequentially execute all the commands in the iterable `commands`.

Returns a Twisted `Deferred` object, whose callback will get a sequence of all the results after the connection is finished.

`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.

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

If any command returns an error, the connection is dropped immediately and the errback will fire with the failed command. You may set `with_errors` to get the exception objects in the list instead.

Connection failures will still fire the errback.

`LoginTimeout` errors are always possible if the login process takes longer than expected and cannot be disabled.

Parameters

- **device** – A `NetDevice` object
- **commands** – An iterable of commands to execute (without newlines).
- **creds** – (Optional) A 2-tuple of (username, password). If unset it will fetch it from `.tacacsrc`.
- **incremental** – (Optional) A callback that will be called with an empty sequence upon connection and then called every time a result comes back from the device, with the list of all results.
- **with_errors** – (Optional) Return exceptions as results instead of raising them
- **timeout** – (Optional) Command response timeout in seconds. Set to `None` to disable. The default is in `settings.DEFAULT_TIMEOUT`. `CommandTimeout` errors will result if a command seems to take longer to return than specified.
- **command_interval** – (Optional) Amount of time in seconds to wait between sending commands.

Returns A Twisted `Deferred` object

trigger.twister.**execute_generic_ssh**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*, *channel=None*, *prompt_pattern=None*, *method='Generic'*)

Use default SSH channel to execute commands on a device. Should work with anything not wonky.

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**execute_ioslike**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*, *loginpw=None*, *enablepw=None*)

Execute commands on a Cisco/IOS-like device. It will automatically try to connect using SSH if it is available and not disabled in `settings.py`. If SSH is unavailable, it will fallback to telnet unless that is also disabled in the settings. Otherwise it will fail, so you should probably make sure one or the other is enabled!

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**execute_ioslike_ssh**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*)

Execute via SSH for IOS-like devices with some exceptions.

Currently confirmed for A10, Brocade MLX, and Cisco only. For all other IOS-like vendors will use telnet for now. :(

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**execute_ioslike_telnet**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*, *loginpw=None*, *enablepw=None*)

Execute commands via telnet on a Cisco/IOS-like device.

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**execute_junoscript**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*)

Connect to a Juniper device and enable Junoscript XML mode. All commands are expected to be XML commands (ElementTree.Element objects suitable for wrapping in `<rpc>` elements). Errors are expected to be of type `xnm:error`. Note that prompt detection is not used here.

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**execute_netscaler**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*)

Execute commands on a NetScaler device.

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**execute_netscreen**(*device*, *commands*, *creds=None*, *incremental=None*, *with_errors=False*, *timeout=300*, *command_interval=0*)

Execute commands on a NetScreen device running ScreenOS. For NetScreen devices running Junos, use `execute_junoscript`.

Please see `execute` for a full description of the arguments and how this works.

trigger.twister.**handle_login_failure**(*failure*)

An errback to try detect a login failure

Parameters **failure** – A Twisted `Failure` instance

trigger.twister.**has_ioslike_error**(*s*)

Test whether a string seems to contain an IOS-like 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 Twisted `Protocol` instance (not class) that will be activated when the session is ready.
- **creds** – A 2-tuple (username, password). By default, credentials from `.tacacsrc` will be used according to `settings.DEFAULT_REALM`. Override that here.
- **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.
- **ping_test** – If set, the device is pinged and must succeed in order to proceed.
- **init_commands** – A list of commands to execute upon logging into the device.

Returns A Twisted `Deferred` object

trigger.twister.**stop_reactor**()
Stop the reactor if it's already running.

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

A collection of CLI tools and utilities used by Trigger.

trigger.utils.**crypt_md5**(*passwd*)
Returns an md5-crypt hash of a clear-text password.

To get md5-crypt from `crypt(3)` you must pass an 8-char string starting with '1' and ending with '$', resulting in a 12-char salt. This only works on systems where md5-crypt is default and is currently assumed to be Linux.

Parameters **passwd** – Password string to be encrypted

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.

The `default` argument is ignored if `autoyes` is set.

Parameters

- **prompt** – Prompt text

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

Default behavior (hitting "enter" returns `False`):

```
>>> yesno('Blow up the moon?')
Blow up the moon? (y/N)
False
```

Reversed behavior (hitting "enter" returns `True`):

```
>>> yesno('Blow up the moon?', default=True)
Blow up the moon? (Y/n)
True
```

Automatically return `True` with `autoyes`; no prompt is displayed:

```
>>> yesno('Blow up the moon?', autoyes=True)
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:

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

`run``()`
Executes the whirlygig!

class `trigger.utils.cli.`**`NullDevice`**
Used to supress output to `sys.stdout` (aka `print`).

Example:

```
>>> from trigger.utils.cli import NullDevice
>>> import sys
>>> print "1 - this will print to STDOUT"
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"
3 - this will print to SDTDOUT
```

trigger.utils.cli.**print_severed_head**()
Prints a demon holding a severed head. Best used when things go wrong, like production-impacting network outages caused by fat-fingered ACL changes.

Thanks to Jeff Sullivan for this best error message ever.

trigger.utils.cli.**min_sec**(*secs*)
Takes an epoch timestamp and returns string of minutes:seconds.

Parameters **secs** – Timestamp (in seconds)

```
>>> import time
>>> start = time.time()  # Wait a few seconds
>>> finish = time.time()
>>> min_sec(finish - start)
'0:11'
```

trigger.utils.cli.**pretty_time**(*t*)
Print a pretty version of timestamp, including timezone info. Expects the incoming datetime object to have proper tzinfo.

Parameters **t** – A `datetime.datetime` object

```
>>> 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.importlib

Utils to import modules.

Taken verbatim from `django.utils.importlib` in Django 1.4.

trigger.utils.importlib.**import_module**(*name*, *package=None*)
Import a module and return the module object.

The `package` argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

trigger.utils.importlib.**import_module_from_path**(*full_path*, *global_name*)
Import a module from a file path and return the module object.

Allows one to import from anywhere, something `__import__()` does not do. The module is added to `sys.modules` as `global_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.

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*, *check_result=False*, *expected_result=''*)

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

If `check_result` is set, the first line of output is retreived from the connection and the starting characters must match `expected_result`.

Parameters

- **host** – Hostname or address
- **port** – Destination port
- **timeout** – Timeout in seconds
- **check_result** – Whether or not to do a string check (e.g. version banner)
- **expected_result** – The expected result!

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

trigger.utils.network.**test_ssh**(*host*, *port=22*, *timeout=5*, *version=('SSH-1.99', 'SSH-2.0')*)

Connect to a TCP port and confirm the SSH version. Defaults to SSHv2.

Note that the default of ('SSH-1.99', 'SSH-2.0') both indicate SSHv2 per RFC 4253. (Ref: http://en.wikipedia.org/wiki/Secure_Shell#Version_1.99)

Parameters

- **host** – Hostname or address
- **port** – Destination port
- **timeout** – Timeout in seconds
- **version** – The SSH version prefix (e.g. "SSH-2.0"). This may also be a tuple of prefixes.

```
>>> test_ssh('localhost')
True
>>> test_ssh('localhost', version='SSH-1.5')
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.

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

trigger.utils.notifications

Pluggable event notification system for Trigger.

trigger.utils.notifications.**send_email**(*addresses*, *subject*, *body*, *sender*, *mailhost='localhost'*)

Sends an email to a list of recipients. Returns `True` when done.

Parameters

- **addresses** – List of email recipients
- **subject** – The email subject
- **body** – The email body
- **sender** – The email sender
- **mailhost** – (Optional) Mail server address

trigger.utils.notifications.**send_notification**(**args*, ***kwargs*)

Simple entry point into `notify` that takes any arguments and tries to handle them to send a notification.

This relies on handlers to be definied within `settings.NOTIFICATION_HANDLERS`.

trigger.utils.notifications.**email_handler**(**args*, ***kwargs*)

Default email notification handler.

trigger.utils.notifications.**notify**(**args*, ***kwargs*)

Iterate thru registered handlers to handle events and send notifications.

Handlers should return `True` if they have performed the desired action or `None` if they have not.

trigger.utils.notifications.handlers

Handlers for event notifications.

Handlers are specified by full module path within `settings.NOTIFICATION_HANDLERS`. These are then imported and registered internally in this module.

The primary public interface to this module is `notify` which is in turn called by `send_notification` to send notifications.

Handlers should return `True` if they have performed the desired action or `None` if they have not.

A handler can either define its own custom behavior, or leverage a custom `Event` object. The goal was to provide a simple public interface to customizing event notifications.

If not customized within NOTIFICATION_HANDLERS, the default notification type is an EmailEvent that is handled by email_handler.

trigger.utils.notifications.handlers.**email_handler**(**args*, ***kwargs*)
Default email notification handler.

trigger.utils.notifications.handlers.**notify**(**args*, ***kwargs*)
Iterate thru registered handlers to handle events and send notifications.

Handlers should return True if they have performed the desired action or None if they have not.

trigger.utils.notifications.events

Event objects for the notification system.

These are intended to be used within event handlers such as email_handler().

If not customized within NOTIFICATION_HANDLERS, the default notification type is an EmailEvent that is handled by email_handler.

class trigger.utils.notifications.events.**Event**(***kwargs*)
Base class for events.

It just populates the attribute dict with all keyword arguments thrown at the constructor.

All Event objects are expected to have a .handle() method that willl be called by a handler function. Any user-defined event objects must have a working .handle() method that returns True upon success or None upon a failure when handling the event passed to it.

If you specify required_args, these must have a value other than None when passed to the constructor.

class trigger.utils.notifications.events.**Notification**(*title=None*, *message=None*, *sender=None*, *recipients=None*, *event_status='failure'*, ***kwargs*)
Base class for notification events.

The title and message arguments are the only two that are required. This is to simplify the interface when sending notifications and will cause notifications to send from the default sender to the default ``recipients that are specified withing the global settings.

If sender or recipients are specified, they will override the global defaults.

Note that this base class has no .handle() method defined.

Parameters

- **title** – The title/subject of the notification
- **message** – The message/body of the notification
- **sender** – A string representing the sender of the notification (such as an email address or a hostname)
- **recipients** – An iterable containing strings representing the recipients of of the notification (such as a list of emails or hostnames)
- **event_status** – Whether this event is a failure or a success

class trigger.utils.notifications.events.**EmailEvent**(*title=None*, *message=None*, *sender=None*, *recipients=None*, *event_status='failure'*, ***kwargs*)
An email notification event.

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
```

6.10 Change Log

Please see the *Changelog*.

6.11 Road Map

We are using milestones to track Trigger's development path 30 to 90 days out. This is where we map outstanding issues to upcoming releases and is the best way to see what's coming!

CHAPTER

SEVEN

DEVELOPMENT

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

CHAPTER

EIGHT

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!

8.1 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!

8.2 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!").

8.3 Email

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

8.4 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.

8.5 IRC

Find us on IRC at `#trigger` on Freenode (`irc://irc.freenode.net`). Trigger is a Pacific coast operation, so your best chance of getting a real-time response is during the weekdays, Pacific time.

8.6 Wiki

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

8.7 OpenHatch

Find Trigger on Openhatch at http://openhatch.org/+projects/Trigger!

CHAPTER

NINE

LICENSE

Trigger is licensed under the BSD 3-Clause License. For the explicit details, please see the *License* page.

CHAPTER

TEN

ABOUT

Trigger was created by AOL's Network Engineering team. With the high number of network devices on the AOL network this application is invaluable to performance and reliability. Hopefully you'll find it useful for it on your network and consider participating!

To learn about Trigger's background and history as well as an overview of the various components, please see the *Overview*.

CHAPTER

ELEVEN

INDICES AND TABLES

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

PYTHON MODULE INDEX

t