Fail2Ban Developers' Documentation

Release 0.9.0.dev

June 22, 2014

Contents

Contents:

CHAPTER 1

How to develop for Fail2Ban

Fail2Ban uses GIT (http://git-scm.com/) distributed source control. This gives each developer their own complete copy of the entire repository. Developers can add and switch branches and commit changes when ever they want and then ask a maintainer to merge their changes.

Fail2Ban uses GitHub (https://github.com/fail2ban/fail2ban) to manage access to the Git repository. GitHub provides free hosting for open-source projects as well as a web-based Git repository browser and an issue tracker.

If you are familiar with Python and you have a bug fix or a feature that you would like to add to Fail2Ban, the best way to do so it to use the GitHub Pull Request feature. You can find more details on the Fail2Ban wiki (http://www.fail2ban.org/wiki/index.php/Get_Involved)

1.1 Pull Requests

When submitting pull requests on GitHub we ask you to:

- Clearly describe the problem you're solving;
- Don't introduce regressions that will make it hard for systems administrators to update;
- If adding a major feature rebase your changes on master and get to a single commit;
- Include test cases (see below);
- Include sample logs (if relevant);
- Include a change to the relevant section of the ChangeLog; and
- Include yourself in THANKS if not already there.

If you are developing filters see the FILTERS file for documentation.

1.2 Code Testing

Existing tests can be run by executing *bin/fail2ban-testcases*. It has options like –log-level that will probably be useful. Run *bin/fail2ban-testcases –help* for the full list of options.

Test cases should cover all usual cases, all exception cases and all inside / outside boundary conditions.

Test cases should cover all branches. The coverage tool will help identify missing branches. Also see http://nedbatchelder.com/code/coverage/branch.html for more details.

Install the package python-coverage to visualise your test coverage. Run the following (note: on Debian-based systems, the script is called *python-coverage*):

```
coverage run bin/fail2ban-testcases
coverage html
```

Then look at htmlcov/index.html and see how much coverage your test cases exert over the code base. Full coverage is a good thing however it may not be complete. Try to ensure tests cover as many independent paths through the code.

Manual Execution. To run in a development environment do:

```
./fail2ban-client -c config/ -s /tmp/f2b.sock -i start
```

some quick commands:

```
status
add test pyinotify
status test
set test addaction iptables
set test actionban iptables echo <ip> <cidr> >> /tmp/ban
set test actionunban iptables echo <ip> <cidr> >> /tmp/unban
get test actionban iptables
get test actionunban iptables
set test banip 192.168.2.2
status test
```

1.3 Coding Standards

1.3.1 Style

Please use tabs for now. Keep to 80 columns, at least for readable text.

1.3.2 Tests

Add tests. They should test all the code you add in a meaning way.

1.3.3 Coverage

Test coverage should always increase as you add code.

You may use "# pragma: no cover" in the code for branches of code that support older versions on python. For all other uses of "pragma: no cover" or "pragma: no branch" document the reason why its not covered. "I haven't written a test case" isn't a sufficient reason.

1.3.4 pyflakes

pyflakes can be used to find unused imports, and unused, undefined and redefined variables. pyflakes should be run in any python code, including python based actions:

```
pyflakes bin/ config/ fail2ban/
```

1.3.5 Documentation

Ensure this documentation is up to date after changes. Also ensure that the man pages still are accurate. Ensure that there is sufficient documentation for your new features to be used.

1.3.6 Bugs

Remove them and don't add any more.

1.3.7 Git

Use the following tags in your commit messages:

- 'BF:' for bug fixes
- 'DOC:' for documentation fixes
- 'ENH:' for enhancements
- 'TST:' for commits concerning tests only (thus not touching the main code-base)

Multiple tags could be joined with +, e.g. "BF+TST:".

Use the text "closes #333"/"resolves #333 "/"fixes #333" where 333 represents an issue that is closed. Other text and details in link below. See: https://help.github.com/articles/closing-issues-via-commit-messages

If merge resulted in conflicts, clarify what changes were done to corresponding files in the 'Conflicts:' section of the merge commit message. See e.g. https://github.com/fail2ban/fail2ban/commit/f5a8a8ac

1.3.8 Adding Actions

If you add an action.d/*.conf file also add a example in config/jail.conf with enabled=false and maxretry=5 for ssh.

1.4 Design

Fail2Ban was initially developed with Python 2.3 (IIRC). It should still be compatible with Python 2.4 and such compatibility assurance makes code ... old-fashioned in many places (RF-Note). In 0.7 the design went through major re-factoring into client/server, a-thread-per-jail design which made it a bit difficult to follow. Below you can find a sketchy description of the main components of the system to orient yourself better.

1.4.1 server/

Core classes hierarchy (feel welcome to draw a better/more complete one):

```
->   inheritance
+    delegation
*    storage of multiple instances

RF-Note   just a note which might be useful to address while doing RF

JailThread -> Filter -> FileFilter -> {FilterPoll, FilterPyinotify, ...}
              |         * FileContainer
              + FailManager
```

```
              + DateDetector
                          + Jail (provided in __init__) which contains this Filter
                (used for passing tickets from FailManager to Jail's __queue)
Server
  + Jails
     * Jail
       + Filter  (in __filter)
       * tickets (in __queue)
       + Actions (in __action)
         * Action
         + BanManager
```

failmanager.py

FailManager

Keeps track of failures, recorded as 'tickets'. All operations are done via acquiring a lock

FailManagerEmpty(Exception)

raised by FailManager.toBan after reaching the list of tickets (RF-Note: asks to become a generator ;))

filter.py

Filter(JailThread)

Wraps (non-threaded) FailManager (and proxies to it quite a bit), and provides all primary logic for processing new lines, what IPs to ignore, etc

.failManager [FailManager] .dateDetector [DateDetector] .__failRegex [list] .__ignoreRegex [list]

Contains regular expressions for failures and ignores

.__findTime [numeric] Used in *processLineAndAdd* to skip old lines

FileFilter(Filter):

Files-aware Filter

.__logPath [list] keeps the tracked files (added 1-by-1 using addLogPath) stored as FileContainer's

.getFailures actually just returns True

if managed to open and get lines (until empty)

False if failed to open or absent container matching the filename

FileContainer

Adapter for a file to deal with log rotation.

.open,.close,.readline RF-Note: readline returns "" with handler absent... shouldn't it be None?

.__pos Keeps the position pointer

dnsutils.py

DNSUtils

Utility class for DNS and IP handling

filter*.py

Implementations of FileFilter's for specific backends. Derived classes should provide an implementation of *run* and usually override *addLogPath*, *delLogPath* methods. In run() method they all one way or another provide

try:

while True: ticket = self.failManager.toBan() self.jail.putFailTicket(ticket)

except FailManagerEmpty: self.failManager.cleanup(MyTime.time())

thus channelling "ban tickets" from their failManager to the corresponding jail.

action.py

Takes care about executing start/check/ban/unban/stop commands

CHAPTER 2

Developing Filters

Filters are tricky. They need to:

- work with a variety of the versions of the software that generates the logs;
- work with the range of logging configuration options available in the software;
- work with multiple operating systems;
- not make assumptions about the log format in excess of the software (e.g. do not assume a username doesn't contain spaces and use S+ unless you've checked the source code);
- account for how future versions of the software will log messages (e.g. guess what would happen to the log message if different authentication types are added);
- not be susceptible to DoS vulnerabilities (see Filter Security below); and
- match intended log lines only.

Please follow the steps from Filter Test Cases to Developing Filter Regular Expressions and submit a GitHub pull request (PR) afterwards. If you get stuck, you can push your unfinished changes and still submit a PR – describe what you have done, what is the hurdle, and we'll attempt to help (PR will be automagically updated with future commits you would push to complete it).

2.1 Filter Test Cases

2.1.1 Purpose

Start by finding the log messages that the application generates related to some form of authentication failure. If you are adding to an existing filter think about whether the log messages are of a similar importance and purpose to the existing filter. If you were a user of Fail2Ban, and did a package update of Fail2Ban that started matching new log messages, would anything unexpected happen? Would the bantime/findtime for the jail be appropriate for the new log messages? If it doesn't, perhaps it needs to be in a separate filter definition, for example like exim filter aims at authentication failures and exim-spam at log messages related to spam.

Even if it is a new filter you may consider separating the log messages into different filters based on purpose.

2.1.2 Cause

Are some of the log lines a result of the same action? For example, is a PAM failure log message, followed by an application specific failure message the result of the same user/script action? If you add regular expressions for both you would end up with two failures for a single action. Therefore, select the most appropriate log message and

document the other log message) with a test case not to match it and a description as to why you chose one over another.

With the selected log lines consider what action has caused those log messages and whether they could have been generated by accident? Could the log message be occurring due to the first step towards the application asking for authentication? Could the log messages occur often? If some of these are true make a note of this in the jail.conf example that you provide.

2.1.3 Samples

It is important to include log file samples so any future change in the regular expression will still work with the log lines you have identified.

The sample log messages are provided in a file under testcases/files/logs/ named identically as the corresponding filter (but without .conf extension). Each log line should be preceded by a line with failJSON metadata (so the logs lines are tested in the test suite) directly above the log line. If there is any specific information about the log message, such as version or an application configuration option that is needed for the message to occur, include this in a comment (line beginning with #) above the failJSON metadata.

Log samples should include only one, definitely not more than 3, examples of log messages of the same form. If log messages are different in different versions of the application log messages that show this are encouraged.

Also attempt to inject an IP into the application (e.g. by specifying it as a username) so that Fail2Ban possibly detects the IP from user input rather than the true origin. See the Filter Security section and the top example in testcases/files/logs/apache-auth as to how to do this. One you have discovered that this is possible, correct the regex so it doesn't match and provide this as a test case with "match": false (see failJSON below).

If the mechanism to create the log message isn't obvious provide a configuration and/or sample scripts testcases/files/config/{filtername} and reference these in the comments above the log line.

2.1.4 FailJSON metadata

A failJSON metadata is a comment immediately above the log message. It will look like:

```
# failJSON: { "time": "2013-06-10T10:10:59", "match": true , "host": "93.184.216.119" }
```

Time should match the time of the log message. It is in a specific format of Year-Month-Day'T'Hour:minute:Second. If your log message does not include a year, like the example below, the year should be listed as 2005, if before Sun Aug 14 10am UTC, and 2004 if afterwards. Here is an example failJSON line preceding a sample log line:

```
# failJSON: { "time": "2005-03-24T15:25:51", "match": true , "host": "198.51.100.87" }
Mar 24 15:25:51 buffalo1 dropbear[4092]: bad password attempt for 'root' from 198.51.100.87:5543
```

The "host" in failJSON should contain the IP or domain that should be blocked.

For long lines that you do not want to be matched (e.g. from log injection attacks) and any log lines to be excluded (see "Cause" section above), set "match": false in the failJSON and describe the reason in the comment above.

After developing regexes, the following command will test all failJSON metadata against the log lines in all sample log files:

```
./fail2ban-testcases testSampleRegex
```

2.2 Developing Filter Regular Expressions

2.2.1 Date/Time

At the moment, Fail2Ban depends on log lines to have time stamps. That is why before starting to develop failregex, check if your log line format known to Fail2Ban. Copy the time component from the log line and append an IP address to test with following command:

```
./fail2ban-regex "2013-09-19 02:46:12 1.2.3.4" "<HOST>"
```

Output of such command should contain something like:

```
Date template hits:
|- [# of hits] date format
|  [1] Year-Month-Day Hour:Minute:Second
```

Ensure that the template description matches time/date elements in your log line time stamp. If there is no matched format then date template needs to be added to server/datedetector.py. Ensure that a new template is added in the order that more specific matches occur first and that there is no confusion between a Day and a Month.

2.2.2 Filter file

The filter is specified in a config/filter.d/{filtername}.conf file. Filter file can have sections INCLUDES (optional) and Definition as follows:

```
[INCLUDES]

before = common.conf

after = filtername.local

[Definition]

failregex = ....

ignoreregex = ....
```

This is also documented in the man page jail.conf (section 5). Other definitions can be added to make failregex's more readable and maintainable to be used through string Interpolations (see http://docs.python.org/2.7/library/configparser.html)

2.2.3 General rules

Use "before" if you need to include a common set of rules, like syslog or if there is a common set of regexes for multiple filters.

Use "after" if you wish to allow the user to overwrite a set of customisations of the current filter. This file doesn't need to exist.

Try to avoid using ignoreregex mainly for performance reasons. The case when you would use it is if in trying to avoid using it, you end up with an unreadable failregex.

2.2.4 Syslog

If your application logs to syslog you can take advantage of log line prefix definitions present in common.conf. So as a base use:

```
[INCLUDES]

before = common.conf

[Definition]

_daemon = app

failregex = ^%(__prefix_line)s
```

In this example common.conf defines __prefix_line which also contains the _daemon name (in syslog terms the service) you have just specified. _daemon can also be a regex.

For example, to capture following line _daemon should be set to "dovecot":

```
Dec 12 11:19:11 dunnart dovecot: pop3-login: Aborted login (tried to use disabled plaintext auth): ri
```

and then `^%(__prefix_line)s` would match "Dec 12 11:19:11 dunnart dovecot: ". Note it matches the trailing space(s) as well.

2.2.5 Substitutions (AKA string interpolations)

We have used string interpolations in above examples. They are useful for making the regexes more readable, reuse generic patterns in multiple failregex lines, and also to refer definition of regex parts to specific filters or even to the user. General principle is that value of a _name variable replaces occurrences of %(_name)s within the same section or anywhere in the config file if defined in [DEFAULT] section.

2.2.6 Regular Expressions

Regular expressions (failregex, ignoreregex) assume that the date/time has been removed from the log line (this is just how fail2ban works internally ATM).

If the format is like '<date...> error 1.2.3.4 is evil' then you need to match the < at the start so regex should be similar to '^<> <HOST> is evil$' using <HOST> where the IP/domain name appears in the log line.

The following general rules apply to regular expressions:

- ensure regexes start with a ^ and are as restrictive as possible. E.g. do not use .* if d+ is sufficient;
- use functionality of Python regexes defined in the standard Python re library http://docs.python.org/2/library/re.html;
- make regular expressions readable (as much as possible). E.g. (?:...) represents a non-capturing regex but (...) is more readable, thus preferred.

If you have only a basic knowledge of regular repressions we advise to read http://docs.python.org/2/library/re.html first. It doesn't take long and would remind you e.g. which characters you need to escape and which you don't.

2.2.7 Developing/testing a regex

You can develop a regex in a file or using command line depending on your preference. You can also use samples you have already created in the test cases or test them one at a time.

The general tool for testing Fail2Ban regexes is fail2ban-regex. To see how to use it run:

```
./fail2ban-regex --help
```

Take note of -l heavydebug / -l debug and -v as they might be very useful.

Tip: Take a look at the source code of the application you are developing failregex for. You may see optional or extra log messages, or parts there of, that need to form part of your regex. It may also reveal how some parts are constrained and different formats depending on configuration or less common usages.

Tip: For looking through source code - http://sourcecodebrowser.com/ . It has call graphs and can browse different versions.

Tip: Some applications log spaces at the end. If you are not sure add s*$ as the end part of the regex.

If your regex is not matching, http://www.debuggex.com/?flavor=python can help to tune it. fail2ban-regex -D ... will present Debuggex URLs for the regexs and sample log files that you pass into it.

In general use when using regex debuggers for generating fail2ban filters: * use regex from the ./fail2ban-regex output (to ensure all substitutions are done) * replace <HOST> with (?&.ipv4) * make sure that regex type set to Python * for the test data put your log output with the date/time removed

When you have fixed the regex put it back into your filter file.

Please spread the good word about Debuggex - Serge Toarca is kindly continuing its free availability to Open Source developers.

2.2.8 Finishing up

If you've added a new filter, add a new entry in config/jail.conf. The theory here is that a user will create a jail.local with [filtername]nenable=true to enable your jail.

So more specifically in the [filter] section in jail.conf:

- ensure that you have "enabled = false" (users will enable as needed);
- use "filter =" set to your filter name;
- use a typical action to disable ports associated with the application;
- set "logpath" to the usual location of application log file;
- if the default findtime or bantime isn't appropriate to the filter, specify more appropriate choices (possibly with a brief comment line).

Submit github pull request (See "Pull Requests" above) for github.com/fail2ban/fail2ban containing your great work.

2.3 Filter Security

Poor filter regular expressions are susceptible to DoS attacks.

When a remote user has the ability to introduce text that would match filter's failregex, while matching inserted text to the <HOST> part, they have the ability to deny any host they choose.

So the <HOST> part must be anchored on text generated by the application, and not the user, to an extent sufficient to prevent user inserting the entire text matching this or any other failregex.

Ideally filter regex should anchor at the beginning and at the end of log line. However as more applications log at the beginning than the end, anchoring the beginning is more important. If the log file used by the application is shared with other applications, like system logs, ensure the other application that use that log file do not log user generated text at the beginning of the line, or, if they do, ensure the regexes of the filter are sufficient to mitigate the risk of insertion.

2.3.1 Examples of poor filters

1. Too restrictive

We find a log message:

```
Apr-07-13 07:08:36 Invalid command fial2ban from 1.2.3.4
```

We make a failregex:

```
^Invalid command \S+ from <HOST>
```

Now think evil. The user does the command 'blah from 1.2.3.44'

The program diligently logs:

```
Apr-07-13 07:08:36 Invalid command blah from 1.2.3.44 from 1.2.3.4
```

And fail2ban matches 1.2.3.44 as the IP that it ban. A DoS attack was successful.

The fix here is that the command can be anything so .* is appropriate:

```
^Invalid command .* from <HOST>
```

Here the .* will match until the end of the string. Then realise it has more to match, i.e. "from <HOST>" and go back until it find this. Then it will ban 1.2.3.4 correctly. Since the <HOST> is always at the end, end the regex with a $:

```
^Invalid command .* from <HOST>$
```

Note if we'd just had the expression:

```
^Invalid command \S+ from <HOST>$
```

Then provided the user put a space in their command they would have never been banned.

2. Unanchored regex can match other user injected data

From the Apache vulnerability CVE-2013-2178 (original ref: https://vndh.net/note:fail2ban-089-denial-service).

An example bad regex for Apache:

```
failregex = [[]client <HOST>[]] user .* not found
```

Since the user can do a get request on:

```
GET /[client%20192.168.0.1]%20user%20root%20not%20found HTTP/1.0
Host: remote.site
```

Now the log line will be:

```
[Sat Jun 01 02:17:42 2013] [error] [client 192.168.33.1] File does not exist: /srv/http/site/[client
```

As this log line doesn't match other expressions hence it matches the above regex and blocks 192.168.33.1 as a denial of service from the HTTP requester.

3. Over greedy pattern matching

From: https://github.com/fail2ban/fail2ban/pull/426

An example ssh log (simplified):

```
Sep 29 17:15:02 spaceman sshd[12946]: Failed password for user from 127.0.0.1 port 20000 ssh1: ruser
```

As we assume username can include anything including spaces its prudent to put .* here. The remote user can also exist as anything so lets not make assumptions again:

```
failregex = ^%(__prefix_line)sFailed \S+ for .* from <HOST>( port \d*)?( ssh\d+)?(: ruser .*)?$
```

So this works. The problem is if the .* after remote user is injected by the user to be 'from 1.2.3.4'. The resultant log line is:

```
Sep 29 17:15:02 spaceman sshd[12946]: Failed password for user from 127.0.0.1 port 20000 ssh1: ruser
```

Testing with:

```
fail2ban-regex -v 'Sep 29 17:15:02 Failed password for user from 127.0.0.1 port 20000 ssh1: ruser fr
```

Tip: I've removed the bit that matches __prefix_line from the regex and log.

Shows:

```
1) [1] ^ Failed \S+ for .* from <HOST>( port \d*)?( ssh\d+)?(: ruser .*)?$
   1.2.3.4  Sun Sep 29 17:15:02 2013
```

It should of matched 127.0.0.1. So the first greedy part of the greedy regex matched until the end of the string. The was no "from <HOST>" so the regex engine worked backwards from the end of the string until this was matched.

The result was that 1.2.3.4 was matched, injected by the user, and the wrong IP was banned.

The solution here is to make the first .* non-greedy with .*?. Here it matches as little as required and the fail2ban-regex tool shows the output:

```
fail2ban-regex -v 'Sep 29 17:15:02 Failed password for user from 127.0.0.1 port 20000 ssh1: ruser fr

1) [1] ^ Failed \S+ for .*? from <HOST>( port \d*)?( ssh\d+)?(: ruser .*)?$
   127.0.0.1  Sun Sep 29 17:15:02 2013
```

So the general case here is a log line that contains:

```
(fixed_data_1)<HOST>(fixed_data_2)(user_injectable_data)
```

Where the regex that matches fixed_data_1 is gready and matches the entire string, before moving backwards and user_injectable_data can match the entire string.

2.3.2 Another case

ref: https://www.debuggex.com/r/CtAbeKMa2sDBEfA2/0

A webserver logs the following without URL escaping:

```
[error] 2865#0: *66647 user "xyz" was not found in "/file", client: 1.2.3.1, server: www.host.com, re
```

regex:

```
failregex = ^ \[error\] \d+#\d+: \*\d+ user "\S+":? (?:password mismatch|was not found in ".*"), clie
```

The .* matches to the end of the string. Finds that it can't continue to match ", client ... so it moves from the back and find that the user injected web URL:

```
", client: 3.2.1.1, server: fake.com, request: "GET exploited HTTP/3.3", host: "injected.host
```

In this case there is a fixed host: "www.myhost.com" at the end so the solution is to anchor the regex at the end with a $.

If this wasn't the case then first .* needed to be made so it didn't capture beyond <HOST>.

4. Application generates two identical log messages with different meanings

If the application generates the following two messages under different circumstances:

```
client <IP>: authentication failed
client <USER>: authentication failed
```

Then it's obvious that a regex of `^client <HOST>: authentication failed$` will still cause problems if the user can trigger the second log message with a <USER> of 123.1.1.1.

Here there's nothing to do except request/change the application so it logs messages differently.

CHAPTER 3

How to do a release for Fail2Ban

3.1 Preparation

- Check distribution patches and see if they can be included
  - https://apps.fedoraproject.org/packages/fail2ban/sources
  - http://sources.gentoo.org/cgi-bin/viewvc.cgi/gentoo-x86/net-analyzer/fail2ban/
  - http://svnweb.freebsd.org/ports/head/security/py-fail2ban/
  - https://build.opensuse.org/package/show?package=fail2ban&project=openSUSE%3AFactory
  - http://sophie.zarb.org/sources/fail2ban (Mageia)
  - https://trac.macports.org/browser/trunk/dports/security/fail2ban
- Check distribution outstanding bugs
  - https://github.com/fail2ban/fail2ban/issues?sort=updated&state=open
  - http://bugs.debian.org/cgi-bin/pkgreport.cgi?dist=unstable;package=fail2ban
  - https://bugs.launchpad.net/ubuntu/+source/fail2ban
  - http://bugs.sabayon.org/buglist.cgi?quicksearch=net-analyzer%2Ffail2ban
  - https://bugs.archlinux.org/?project=5&cat%5B%5D=33&string=fail2ban
  - https://bugs.gentoo.org/buglist.cgi?query_format=advanced&short_desc=fail2ban&bug_status=UNCONFIRMED&bug_sta
  - https://bugzilla.redhat.com/buglist.cgi?query_format=advanced&bug_status=NEW&bug_status=ASSIGNED&component=
  - http://www.freebsd.org/cgi/query-pr-summary.cgi?text=fail2ban
  - https://bugs.mageia.org/buglist.cgi?quicksearch=fail2ban
  - https://build.opensuse.org/package/requests/openSUSE:Factory/fail2ban
- Make sure the tests pass:

  ```
  ./fail2ban-testcases-all
  ```

- Ensure the version is correct in:
  - ./fail2ban/version.py
  - top of ChangeLog
  - README.md

- Ensure the MANIFEST is complete
- Run:

  python setup.py sdist
- Look for errors like:

  ```
  'testcases/files/logs/mysqld.log' not a regular file -- skipping
  ```

  - Which indicates that testcases/files/logs/mysqld.log has been moved or is a directory:

    ```
    tar -C /tmp -jxf dist/fail2ban-0.9.0.tar.bz2
    ```
- clean up current direcory:

  ```
  diff -rul --exclude \*.pyc . /tmp/fail2ban-0.9.0/
  ```

  - Only differences should be files that you don't want distributed.
- Ensure the tests work from the tarball:

  ```
  cd /tmp/fail2ban-0.9.0/ && export PYTHONPATH=`pwd` && bin/fail2ban-testcases
  ```
- Add/finalize the corresponding entry in the ChangeLog
  - To generate a list of committers use e.g.:

    ```
    git shortlog -sn 0.8.12.. | sed -e 's,^[ 0-9\t]*,,g' | tr '\n' '\|' | sed -e 's:|:, :g'
    ```
  - Ensure the top of the ChangeLog has the right version and current date.
  - Ensure the top entry of the ChangeLog has the right version and current date.
- Update man pages:

  ```
  (cd man ; ./generate-man )
  git commit -m 'DOC/ENH: update man pages for release' man/*
  ```
- Cleanout TODO file with the finished stuff
- Prepare source and rpm binary distributions:

  ```
  python setup.py sdist
  ```

  - Broken for now: python setup.py bdist_rpm
  - Broken for now: python setup.py upload
- Tag the release by using a signed (and annotated) tag. Cut/paste release ChangeLog entry as tag annotation:

  ```
  git tag -s 0.9.1
  ```

3.2 Pre Release

- Provide a release sample to distributors
  - Arch Linux:
    - https://www.archlinux.org/packages/community/any/fail2ban/
  - Debian: Yaroslav Halchenko <debian@onerussian.com>
    - http://packages.qa.debian.org/f/fail2ban.html

  - FreeBSD: Christoph Theis theis@gmx.at>, Nick Hilliard <nick@foobar.org>
    * http://svnweb.freebsd.org/ports/head/security/py-fail2ban/Makefile?view=markup
    * http://www.freebsd.org/cgi/query-pr-summary.cgi?text=fail2ban
  - Fedora: Axel Thimm <Axel.Thimm@atrpms.net>
    * https://apps.fedoraproject.org/packages/fail2ban
    * http://pkgs.fedoraproject.org/cgit/fail2ban.git
    * https://admin.fedoraproject.org/pkgdb/acls/bugs/fail2ban
  - Gentoo: netmon@gentoo.org
    * http://sources.gentoo.org/cgi-bin/viewvc.cgi/gentoo-x86/net-analyzer/fail2ban/metadata.xml?view=markup
    * https://bugs.gentoo.org/buglist.cgi?quicksearch=fail2ban
  - openSUSE: Stephan Kulow <coolo@suse.com>
    * https://build.opensuse.org/package/show/openSUSE:Factory/fail2ban
  - Mac Ports: @Malbrouck on github (gh-49)
    * https://trac.macports.org/browser/trunk/dports/security/fail2ban/Portfile
  - Mageia:
    * https://bugs.mageia.org/buglist.cgi?quicksearch=fail2ban
  - An potentially to the fail2ban-users email list.
- Wait for feedback from distributors
- Prepare a release notice https://github.com/fail2ban/fail2ban/releases/new
- Upload the source/binaries from the dist directory and tag the release using the URL
- Upload source/binaries to sourceforge http://sourceforge.net/projects/fail2ban/
- Run the following and update the wiki with output:

```
python -c 'import fail2ban.protocol; fail2ban.protocol.printWiki()'
```

  - page: http://www.fail2ban.org/wiki/index.php/Commands
- Update:
  - http://www.fail2ban.org/wiki/index.php?title=Template:Fail2ban_Versions&action=edit
  - http://www.fail2ban.org/wiki/index.php?title=Template:Fail2ban_News&action=edit * move old bits to http://www.fail2ban.org/wiki/index.php?title=Template:Fail2ban_OldNews&action=edit
  - http://www.fail2ban.org/wiki/index.php/ChangeLog
  - http://www.fail2ban.org/wiki/index.php/Requirements (Check requirement)
  - http://www.fail2ban.org/wiki/index.php/Features
- See if any filters are upgraded: http://www.fail2ban.org/wiki/index.php/Special:AllPages
- Email users and development list of release
- notify distributors

3.3 Post Release

Add the following to the top of the ChangeLog:

```
ver. 0.9.1 (2014/XX/XXX) - wanna-be-released
-----------

- Fixes:

- New Features:

- Enhancements:
```

Alter the git shortlog command in the previous section to refer to the just released version.

and adjust fail2ban/version.py to carry .dev suffix to signal a version under development.

CHAPTER 4

fail2ban package

4.1 fail2ban.client package

4.1.1 fail2ban.client.actionreader module

class fail2ban.client.actionreader.**ActionReader**(*file_*, *jailName*, *initOpts*, ***kwargs*)
Bases: fail2ban.client.configreader.DefinitionInitConfigReader

Methods

add_section(section)	Create a new section in the configuration.
convert()	
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getFile()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getJailName()	
getName()	
getOptions(pOpts)	
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.
optionxform(optionstr)	
read()	
readexplicit()	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	
setFile(fileName)	

Continued on next page

Table 4.1 – continued from previous page

setJailName(jailName)	
setName(name)	
write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

convert()

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

getFile()

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getJailName()

getName()

getOptions(*pOpts*)

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items (*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

options (*section*)
Return a list of option names for the given section name.

optionxform (*optionstr*)

read ()

readexplicit ()

readfp (*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option (*section*, *option*)
Remove an option.

remove_section (*section*)
Remove a file section.

sections ()
Return a list of section names, excluding [DEFAULT]

set (*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir (*basedir*)

setFile (*fileName*)

setJailName (*jailName*)

setName (*name*)

write (*fp*)
Write an .ini-format representation of the configuration state.

4.1.2 fail2ban.client.beautifier module

class fail2ban.client.beautifier.**Beautifier** (*cmd=None*)

Methods

beautify(response)	
beautifyError(response)	
getInputCmd()	
setInputCmd(cmd)	

beautify (*response*)

beautifyError(*response*)

getInputCmd()

setInputCmd(*cmd*)

4.1.3 fail2ban.client.configparserinc module

class fail2ban.client.configparserinc.**SafeConfigParserWithIncludes**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*)

Bases: ConfigParser.SafeConfigParser

Class adds functionality to SafeConfigParser to handle included other configuration files (or may be urls, whatever in the future)

File should have section [includes] and only 2 options implemented are 'files_before' and 'files_after' where files are listed 1 per line.

Example:

[INCLUDES] before = 1.conf

3.conf

after = 1.conf

It is a simple implementation, so just basic care is taken about recursion. Includes preserve right order, ie new files are inserted to the list of read configs before original, and their includes correspondingly so the list should follow the leaves of the tree.

I wasn't sure what would be the right way to implement generic (aka c++ template) so we could base at any *configparser class... so I will leave it for the future

Methods

add_section(section)	Create a new section in the configuration.
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.
optionxform(optionstr)	
read(filenames)	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]

Continued on next page

Table 4.3 – continued from previous page

set(section, option[, value])	Set an option.
write(fp)	Write an .ini-format representation of the configuration state.

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items(*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

options(*section*)
Return a list of option names for the given section name.

optionxform(*optionstr*)

read(*filenames*)

readfp(*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option(*section*, *option*)
Remove an option.

remove_section(*section*)
Remove a file section.

sections()
Return a list of section names, excluding [DEFAULT]

set(*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

write(*fp*)
Write an .ini-format representation of the configuration state.

4.1.4 fail2ban.client.configreader module

class fail2ban.client.configreader.**ConfigReader**(*basedir=None*)
Bases: fail2ban.client.configparserinc.SafeConfigParserWithIncludes

Methods

add_section(section)	Create a new section in the configuration.
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getOptions(sec, options[, pOptions])	
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.
optionxform(optionstr)	
read(filename)	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	
write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getOptions(*sec*, *options*, *pOptions=None*)

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items(*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

options(*section*)
Return a list of option names for the given section name.

optionxform(*optionstr*)

read(*filename*)

readfp(*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option(*section*, *option*)
Remove an option.

remove_section(*section*)
Remove a file section.

sections()
Return a list of section names, excluding [DEFAULT]

set(*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir(*basedir*)

write(*fp*)
Write an .ini-format representation of the configuration state.

class fail2ban.client.configreader.**DefinitionInitConfigReader**(*file_*, *jailName*, *initOpts*, ***kwargs*)
Bases: fail2ban.client.configreader.ConfigReader

Config reader for files with options grouped in [Definition] and [Init] sections.

Is a base class for readers of filters and actions, where definitions in jails might provide custom values for options defined in [Init] section.

Methods

add_section(section)	Create a new section in the configuration.
convert()	
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getFile()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getJailName()	
getOptions(pOpts)	
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.
optionxform(optionstr)	
read()	
readexplicit()	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	

Continued on next page

Table 4.5 – continued from previous page

setFile(fileName)	
setJailName(jailName)	
write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

convert()

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

getFile()

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getJailName()

getOptions(*pOpts*)

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items(*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

options(*section*)
Return a list of option names for the given section name.

optionxform(*optionstr*)

read()

readexplicit()

readfp(*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option(*section*, *option*)
Remove an option.

remove_section(*section*)
Remove a file section.

sections()
Return a list of section names, excluding [DEFAULT]

set(*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir(*basedir*)

setFile(*fileName*)

setJailName(*jailName*)

write(*fp*)
Write an .ini-format representation of the configuration state.

4.1.5 fail2ban.client.configurator module

class fail2ban.client.configurator.**Configurator**

Methods

	convertToProtocol()
	getBaseDir()
	getConfigStream()
	getEarlyOptions()
	getOptions([jail])
	readAll()
	readEarly()
	setBaseDir(folderName)

convertToProtocol()

getBaseDir()

getConfigStream()

getEarlyOptions()

getOptions(*jail=None*)

readAll()

readEarly()

setBaseDir(*folderName*)

4.1.6 fail2ban.client.csocket module

class fail2ban.client.csocket.**CSocket**(*sock='/var/run/fail2ban/fail2ban.sock'*)

Methods

receive(sock)	
send(msg)	

END_STRING = '<F2B_END_COMMAND>'

static receive(*sock*)

send(*msg*)

4.1.7 fail2ban.client.fail2banreader module

class fail2ban.client.fail2banreader.**Fail2banReader**(***kwargs*)
Bases: fail2ban.client.configreader.ConfigReader

Methods

add_section(section)	Create a new section in the configuration.
convert()	
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getEarlyOptions()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getOptions()	
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.

Continued on next page

Table 4.8 – continued from previous page

optionxform(optionstr)	
read()	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	
write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

convert()

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

getEarlyOptions()

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getOptions()

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items (*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

options (*section*)
Return a list of option names for the given section name.

optionxform (*optionstr*)

read ()

readfp (*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option (*section*, *option*)
Remove an option.

remove_section (*section*)
Remove a file section.

sections ()
Return a list of section names, excluding [DEFAULT]

set (*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir (*basedir*)

write (*fp*)
Write an .ini-format representation of the configuration state.

4.1.8 fail2ban.client.filterreader module

class fail2ban.client.filterreader.**FilterReader** (*file_*, *jailName*, *initOpts*, ***kwargs*)
Bases: fail2ban.client.configreader.DefinitionInitConfigReader

Methods

add_section(section)	Create a new section in the configuration.
convert()	
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getFile()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getJailName()	
getOptions(pOpts)	
getboolean(section, option)	

Continued on next page

Table 4.9 – continued from previous page

getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.
optionxform(optionstr)	
read()	
readexplicit()	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	
setFile(fileName)	
setJailName(jailName)	
write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

convert()

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

getFile()

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getJailName()

getOptions(*pOpts*)

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items(*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

options(*section*)
Return a list of option names for the given section name.

optionxform(*optionstr*)

read()

readexplicit()

readfp(*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option(*section*, *option*)
Remove an option.

remove_section(*section*)
Remove a file section.

sections()
Return a list of section names, excluding [DEFAULT]

set(*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir(*basedir*)

setFile(*fileName*)

setJailName(*jailName*)

write(*fp*)
Write an .ini-format representation of the configuration state.

4.1.9 fail2ban.client.jailreader module

class fail2ban.client.jailreader.**JailReader**(*name*, *force_enable=False*, ***kwargs*)
Bases: fail2ban.client.configreader.ConfigReader

Attributes

options

Methods

add_section(section)	Create a new section in the configuration.
convert([allow_no_files])	Convert read before __opts to the commands stream
defaults()	
extractOptions(option)	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getName()	
getOptions()	
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
isEnabled()	
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
optionxform(optionstr)	
read()	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	
setName(value)	
write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

convert(*allow_no_files=False*)
Convert read before __opts to the commands stream

Parameters **allow_missing** : bool

Either to allow log files to be missing entirely. Primarily is used for testing

defaults()

static extractOptions(*option*)

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

static getIncludes(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getName()

getOptions()

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

isEnabled()

items(*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

optionCRE = <_sre.SRE_Pattern object at 0x7f42ff0f4da0>

optionExtractRE = <_sre.SRE_Pattern object at 0x7f42ff639750>

options

optionxform(*optionstr*)

read()

readfp(*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option(*section*, *option*)
Remove an option.

remove_section(*section*)
Remove a file section.

sections()
Return a list of section names, excluding [DEFAULT]

set(*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir(*basedir*)

setName(*value*)

write(*fp*)
Write an .ini-format representation of the configuration state.

4.1.10 fail2ban.client.jailsreader module

class fail2ban.client.jailsreader.**JailsReader**(*force_enable=False*, ***kwargs*)
Bases: fail2ban.client.configreader.ConfigReader

Attributes

jails

Methods

add_section(section)	Create a new section in the configuration.
convert([allow_no_files])	Convert read before __opts and jails to the commands stream
defaults()	
get(section, option[, raw, vars])	Get an option value for a given section.
getBaseDir()	
getIncludes(resource[, seen])	Given 1 config resource returns list of included files
getOptions([section])	Reads configuration for jail(s) and adds enabled jails to __jails
getboolean(section, option)	
getfloat(section, option)	
getint(section, option)	
has_option(section, option)	Check for the existence of a given option in a given section.
has_section(section)	Indicate whether the named section is present in the configuration.
items(section[, raw, vars])	Return a list of tuples with (name, value) for each option in the section.
options(section)	Return a list of option names for the given section name.
optionxform(optionstr)	
read()	
readfp(fp[, filename])	Like read() but the argument must be a file-like object.
remove_option(section, option)	Remove an option.
remove_section(section)	Remove a file section.
sections()	Return a list of section names, excluding [DEFAULT]
set(section, option[, value])	Set an option.
setBaseDir(basedir)	

Continued on next page

Table 4.13 – continued from previous page

write(fp)	Write an .ini-format representation of the configuration state.

DEFAULT_BASEDIR = '/etc/fail2ban'

OPTCRE = <_sre.SRE_Pattern object at 0x7f4302325960>

OPTCRE_NV = <_sre.SRE_Pattern object at 0x7f4302194510>

SECTCRE = <_sre.SRE_Pattern object at 0x7f4301ed91b0>

SECTION_NAME = 'INCLUDES'

add_section(*section*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

convert(*allow_no_files=False*)
Convert read before __opts and jails to the commands stream

Parameters **allow_missing** : bool

Either to allow log files to be missing entirely. Primarily is used for testing

defaults()

get(*section*, *option*, *raw=False*, *vars=None*)
Get an option value for a given section.

If 'vars' is provided, it must be a dictionary. The option is looked up in 'vars' (if provided), 'section', and in 'defaults' in that order.

All % interpolations are expanded in the return values, unless the optional argument 'raw' is true. Values for interpolation keys are looked up in the same manner as the option.

The section DEFAULT is special.

getBaseDir()

static **getIncludes**(*resource*, *seen=[]*)
Given 1 config resource returns list of included files (recursively) with the original one as well Simple loops are taken care about

getOptions(*section=None*)
Reads configuration for jail(s) and adds enabled jails to __jails

getboolean(*section*, *option*)

getfloat(*section*, *option*)

getint(*section*, *option*)

has_option(*section*, *option*)
Check for the existence of a given option in a given section.

has_section(*section*)
Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

items(*section*, *raw=False*, *vars=None*)
Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument 'raw' is true. Additional substitutions may be provided using the 'vars' argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

jails

options(*section*)
Return a list of option names for the given section name.

optionxform(*optionstr*)

read()

readfp(*fp*, *filename=None*)
Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<???>' is used.

remove_option(*section*, *option*)
Remove an option.

remove_section(*section*)
Remove a file section.

sections()
Return a list of section names, excluding [DEFAULT]

set(*section*, *option*, *value=None*)
Set an option. Extend ConfigParser.set: check for string values.

setBaseDir(*basedir*)

write(*fp*)
Write an .ini-format representation of the configuration state.

4.2 fail2ban.server package

4.2.1 fail2ban.server.action module

class fail2ban.server.action.**ActionBase**(*jail*, *name*)
Bases: object

An abstract base class for actions in Fail2Ban.

Action Base is a base definition of what methods need to be in place to create a Python based action for Fail2Ban. This class can be inherited from to ease implementation. Required methods:

- __init__(jail, name)
- start()
- stop()
- ban(aInfo)
- unban(aInfo)

Called when action is created, but before the jail/actions is started. This should carry out necessary methods to initialise the action but not "start" the action.

Parameters **jail** : Jail

The jail in which the action belongs to.

name : str

Name assigned to the action.

Notes

Any additional arguments specified in *jail.conf* or passed via *fail2ban-client* will be passed as keyword arguments.

Methods

ban(aInfo)	Executed when a ban occurs.
start()	Executed when the jail/action is started.
stop()	Executed when the jail/action is stopped.
unban(aInfo)	Executed when a ban expires.

ban(*aInfo*)
Executed when a ban occurs.

Parameters **aInfo** : dict

Dictionary which includes information in relation to the ban.

start()
Executed when the jail/action is started.

stop()
Executed when the jail/action is stopped.

unban(*aInfo*)
Executed when a ban expires.

Parameters **aInfo** : dict

Dictionary which includes information in relation to the ban.

class fail2ban.server.action.**CallingMap**(**args*, ***kwargs*)
Bases: _abcoll.MutableMapping

A Mapping type which returns the result of callable values.

CallingMap behaves similar to a standard python dictionary, with the exception that any values which are callable, are called and the result is returned as the value. No error handling is in place, such that any errors raised in the callable will raised as usual. Actual dictionary is stored in property *data*, and can be accessed to obtain original callable values.

Attributes

data	(dict) The dictionary data which can be accessed to obtain items uncalled

Methods

clear(() -> None. Remove all items from D.)	
get((k[,d]) -> D[k] if k in D, ...)	
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem(() -> (k, v), ...)	as a 2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update(([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
values(() -> list of D's values)	

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

class fail2ban.server.action.**CommandAction**(*jail*, *name*)
Bases: fail2ban.server.action.ActionBase

A action which executes OS shell commands.

This is the default type of action which Fail2Ban uses.

Default sets all commands for actions as empty string, such no command is executed.

Parameters **jail** : Jail

The jail in which the action belongs to.

name : str

Name assigned to the action.

Attributes

actionban	The command used when a ban occurs.
actionstart	The command executed on start of the jail/action.
actionstop	The command executed when the jail/actions stops.
actionunban	The command used when an unban occurs.
timeout	Time out period in seconds for execution of commands.

Methods

ban(aInfo)	Executes the "actionban" command.
escapeTag(value)	Escape characters which may be used for command injection.
executeCmd(realCmd[, timeout])	Executes a command.
replaceTag(query, aInfo)	Replaces tags in *query* with property values.
start()	Executes the "actionstart" command.
stop()	Executes the "actionstop" command.
substituteRecursiveTags(tags)	Sort out tag definitions within other tags.
unban(aInfo)	Executes the "actionunban" command.

actionban
The command used when a ban occurs.

actioncheck
The command used to check the environment.

This is used prior to a ban taking place to ensure the environment is appropriate. If this check fails, *stop* and *start* is executed prior to the check being called again.

actionstart
The command executed on start of the jail/action.

actionstop
The command executed when the jail/actions stops.

actionunban
The command used when an unban occurs.

ban(*aInfo*)
Executes the "actionban" command.

Replaces the tags in the action command with actions properties and ban information, and executes the resulting command.

Parameters **aInfo** : dict

Dictionary which includes information in relation to the ban.

static **escapeTag**(*value*)
Escape characters which may be used for command injection.

Parameters **value** : str

A string of which characters will be escaped.

Returns str

value with certain characters escaped.

Notes

The following characters are escaped:

```
\#&;`|*?~<>^()[]{}$'"
```

static executeCmd(*realCmd*, *timeout=60*)
Executes a command.

Parameters **realCmd** : str

The command to execute.

timeout : int

The time out in seconds for the command.

Returns bool

True if the command succeeded.

Raises **OSError**

If command fails to be executed.

RuntimeError

If command execution times out.

classmethod replaceTag(*query*, *aInfo*)
Replaces tags in *query* with property values.

Parameters **query** : str

String with tags.

aInfo : dict

Tags(keys) and associated values for substitution in query.

Returns str

query string with tags replaced.

start()
Executes the "actionstart" command.

Replace the tags in the action command with actions properties and executes the resulting command.

stop()
Executes the "actionstop" command.

Replaces the tags in the action command with actions properties and executes the resulting command.

classmethod substituteRecursiveTags(*tags*)
Sort out tag definitions within other tags.

so: becomes: a = 3 a = 3 b = <a>_3 b = 3_3

Parameters **tags** : dict

Dictionary of tags(keys) and their values.

Returns dict

Dictionary of tags(keys) and their values, with tags within the values recursively replaced.

timeout
Time out period in seconds for execution of commands.

unban(*aInfo*)
Executes the "actionunban" command.

Replaces the tags in the action command with actions properties and ban information, and executes the resulting command.

Parameters **aInfo** : dict

Dictionary which includes information in relation to the ban.

4.2.2 fail2ban.server.actions module

class fail2ban.server.actions.**Actions**(*jail*)
Bases: fail2ban.server.jailthread.JailThread, _abcoll.Mapping

Handles jail actions.

This class handles the actions of the jail. Creation, deletion or to actions must be done through this class. This class is based on the Mapping type, and the *add* method must be used to add new actions. This class also starts and stops the actions, and fetches bans from the jail executing these bans via the actions.

Parameters **jail: Jail**

The jail of which the actions belongs to.

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
status	Status of active bans, and total ban counts.

active	(bool) Control the state of the thread.
idle	(bool) Control the idle state of the thread.
sleeptime	(int) The time the thread sleeps for in the loop.

Methods

add(name[, pythonModule, initOpts])	Adds a new action.
get((k[,d]) -> D[k] if k in D, ...)	
getBanTime()	
getName()	
isAlive()	Return whether the thread is alive.
isDaemon()	
is_alive()	Return whether the thread is alive.
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	

Continued on next page

Table 4.19 – continued from previous page

join([timeout])	Wait until the thread terminates.
keys(() -> list of D's keys)	
removeBannedIP(ip)	Removes banned IP calling actions' unban method
run()	Main loop for Threading.
setBanTime(value)	
setDaemon(daemonic)	
setName(name)	
start()	Sets active flag and starts thread.
stop()	Sets *active* property to False, to flag run method to return.
values(() -> list of D's values)	

add(*name*, *pythonModule=None*, *initOpts=None*)
Adds a new action.

Add a new action if not already present, defaulting to standard *CommandAction*, or specified Python module.

Parameters **name** : str

The name of the action.

pythonModule : str, optional

Path to Python file which must contain *Action* class. Default None, which means *CommandAction* is used.

initOpts : dict, optional

Options for Python Action, used as keyword arguments for initialisation. Default None.

Raises **ValueError**

If action name already exists.

RuntimeError

If external Python module does not have *Action* class or does not implement necessary methods as per *ActionBase* abstract class.

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

get($k[, d]$) → D[k] if k in D, else d. d defaults to None.

getBanTime()

getName()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

keys() → list of D's keys

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

removeBannedIP(*ip*)
Removes banned IP calling actions' unban method

Remove a banned IP now, rather than waiting for it to expire, even if set to never expire.

Parameters **ip** : str

The IP address to unban

Raises **ValueError**

If *ip* is not banned

run()
Main loop for Threading.

This function is the main loop of the thread. It checks the jail queue and executes commands when an IP address is banned.

Returns bool

True when the thread exits nicely.

setBanTime(*value*)

setDaemon(*daemonic*)

setName(*name*)

start()
Sets active flag and starts thread.

status
Status of active bans, and total ban counts.

stop()
Sets *active* property to False, to flag run method to return.

values() → list of D's values

4.2.3 fail2ban.server.asyncserver module

class fail2ban.server.asyncserver.**AsyncServer**(*transmitter*)
Bases: asyncore.dispatcher

Attributes

addr	

Methods

	accept()
	add_channel([map])
	bind(addr)
	close()
	connect(address)
	create_socket(family, type)
	del_channel([map])
	handle_accept()
	handle_close()
	handle_connect()
	handle_connect_event()
	handle_error()
	handle_expt()
	handle_expt_event()
	handle_read()
	handle_read_event()
	handle_write()
	handle_write_event()
	listen(num)
	log(message)
	log_info(message[, type])
	readable()
	recv(buffer_size)
	send(data)
	set_reuse_addr()

Continued on next page

Table 4.20 – continued from previous page

set_socket(sock[, map])
start(sock, force)
stop()
writable()

accept()

accepting = False

add_channel(*map=None*)

addr = None

bind(*addr*)

close()

closing = False

connect(*address*)

connected = False

connecting = False

create_socket(*family*, *type*)

debug = False

del_channel(*map=None*)

handle_accept()

handle_close()

handle_connect()

handle_connect_event()

handle_error()

handle_expt()

handle_expt_event()

handle_read()

handle_read_event()

handle_write()

handle_write_event()

ignore_log_types = frozenset(['warning'])

listen(*num*)

log(*message*)

log_info(*message*, *type='info'*)

readable()

recv(*buffer_size*)

send(*data*)

set_reuse_addr()

set_socket(*sock*, *map=None*)

start(*sock*, *force*)

stop()

writable()

exception fail2ban.server.asyncserver.**AsyncServerException**
Bases: exceptions.Exception

args

message

class fail2ban.server.asyncserver.**RequestHandler**(*conn*, *transmitter*)
Bases: asynchat.async_chat

Attributes

addr	

Methods

accept()	
add_channel([map])	
bind(addr)	
close()	
close_when_done()	automatically close this channel once the outgoing queue is empty
collect_incoming_data(data)	
connect(address)	
create_socket(family, type)	
del_channel([map])	
discard_buffers()	
found_terminator()	
get_terminator()	
handle_accept()	
handle_close()	
handle_connect()	
handle_connect_event()	
handle_error()	
handle_expt()	
handle_expt_event()	
handle_read()	
handle_read_event()	
handle_write()	
handle_write_event()	
initiate_send()	
listen(num)	
log(message)	
log_info(message[, type])	
push(data)	
push_with_producer(producer)	
readable()	predicate for inclusion in the readable for select()

Continued on next page

Table 4.21 – continued from previous page

recv(buffer_size)	
send(data)	
set_reuse_addr()	
set_socket(sock[, map])	
set_terminator(term)	Set the input delimiter.
writable()	predicate for inclusion in the writable for select()

END_STRING = '<F2B_END_COMMAND>'

ac_in_buffer_size = 4096

ac_out_buffer_size = 4096

accept()

accepting = False

add_channel(*map=None*)

addr = None

bind(*addr*)

close()

close_when_done()
automatically close this channel once the outgoing queue is empty

closing = False

collect_incoming_data(*data*)

connect(*address*)

connected = False

connecting = False

create_socket(*family*, *type*)

debug = False

del_channel(*map=None*)

discard_buffers()

found_terminator()

get_terminator()

handle_accept()

handle_close()

handle_connect()

handle_connect_event()

handle_error()

handle_expt()

handle_expt_event()

handle_read()

handle_read_event()

handle_write()

handle_write_event()

ignore_log_types = frozenset(['warning'])

initiate_send()

listen(*num*)

log(*message*)

log_info(*message*, *type='info'*)

push(*data*)

push_with_producer(*producer*)

readable()
predicate for inclusion in the readable for select()

recv(*buffer_size*)

send(*data*)

set_reuse_addr()

set_socket(*sock*, *map=None*)

set_terminator(*term*)
Set the input delimiter. Can be a fixed string of any length, an integer, or None

writable()
predicate for inclusion in the writable for select()

4.2.4 fail2ban.server.banmanager module

class fail2ban.server.banmanager.**BanManager**

Methods

addBanTicket(ticket)	
createBanTicket(ticket)	
flushBanList()	
getBanList()	
getBanTime()	
getBanTotal()	
getTicketByIP(ip)	
setBanTime(value)	
setBanTotal(value)	
size()	
unBanList(time)	

addBanTicket(*ticket*)

static createBanTicket(*ticket*)

flushBanList()

getBanList()

getBanTime()

getBanTotal()

getTicketByIP(*ip*)

setBanTime(*value*)

setBanTotal(*value*)

size()

unBanList(*time*)

4.2.5 fail2ban.server.database module

class fail2ban.server.database.**Fail2BanDb**(*filename*, *purgeAge=86400*)
Bases: object

Fail2Ban database for storing persistent data.

This allows after Fail2Ban is restarted to reinstated bans and to continue monitoring logs from the same point.

This will either create a new Fail2Ban database, connect to an existing, and if applicable upgrade the schema in the process.

Parameters **filename** : str

File name for SQLite3 database, which will be created if doesn't already exist.

purgeAge : int

Purge age in seconds, used to remove old bans from database during purge.

Raises **sqlite3.OperationalError**

Error connecting/creating a SQLite3 database.

RuntimeError

If exisiting database fails to update to new schema.

Attributes

filename	File name of SQLite3 database file.
purgeage	Purge age in seconds.

Methods

addBan(*args, **kwargs)	Add a ban to the database.
addJail(*args, **kwargs)	Adds a jail to the database.
addLog(*args, **kwargs)	Adds a log to the database.
createDb(*args, **kwargs)	Creates a new database, called during initialisation.
delAllJails(*args, **kwargs)	Deletes all jails from the database.
delJail(*args, **kwargs)	Deletes a jail from the database.
getBans(**kwargs)	Get bans from the database.
getBansMerged([ip, jail, bantime])	Get bans from the database, merged into single ticket.
getJailNames(*args, **kwargs)	Get name of jails in database.

Continued on next page

Table 4.24 – continued from previous page

getLogPaths(*args, **kwargs)	Gets all the log paths from the database.
purge(*args, **kwargs)	Purge old bans, jails and log files from database.
updateDb(*args, **kwargs)	Update an existing database, called during initialisation.
updateLog(*args, **kwargs)	Updates hash and last position in log file.

addBan(**args*, ***kwargs*)
Add a ban to the database.

Parameters **jail** : Jail

Jail in which the ban has occurred.

ticket : BanTicket

Ticket of the ban to be added.

addJail(**args*, ***kwargs*)
Adds a jail to the database.

Parameters **jail** : Jail

Jail to be added to the database.

addLog(**args*, ***kwargs*)
Adds a log to the database.

Parameters **jail** : Jail

Jail that log is being monitored by.

container : FileContainer

File container of the log file being added.

Returns int

If log was already present in database, value of last position in the log file; else *None*

createDb(**args*, ***kwargs*)
Creates a new database, called during initialisation.

delAllJails(**args*, ***kwargs*)
Deletes all jails from the database.

delJail(**args*, ***kwargs*)
Deletes a jail from the database.

Parameters **jail** : Jail

Jail to be removed from the database.

filename
File name of SQLite3 database file.

getBans(***kwargs*)
Get bans from the database.

Parameters **jail** : Jail

Jail that the ban belongs to. Default *None*; all jails.

bantime : int

Ban time in seconds, such that bans returned would still be valid now. Negative values are equivalent to *None*. Default *None*; no limit.

ip : str

IP Address to filter bans by. Default *None*; all IPs.

Returns list

List of 'Ticket's for bans stored in database.

getBansMerged(*ip=None*, *jail=None*, *bantime=None*)
Get bans from the database, merged into single ticket.

This is the same as *getBans*, but bans merged into single ticket.

Parameters **jail** : Jail

Jail that the ban belongs to. Default *None*; all jails.

bantime : int

Ban time in seconds, such that bans returned would still be valid now. Negative values are equivalent to *None*. Default *None*; no limit.

ip : str

IP Address to filter bans by. Default *None*; all IPs.

Returns list or Ticket

Single ticket representing bans stored in database per IP in a list. When *ip* argument passed, a single *Ticket* is returned.

getJailNames(**args*, ***kwargs*)
Get name of jails in database.

Currently only used for testing purposes.

Returns set

Set of jail names.

getLogPaths(**args*, ***kwargs*)
Gets all the log paths from the database.

Currently only for testing purposes.

Parameters **jail** : Jail

If specified, will only reutrn logs belonging to the jail.

Returns set

Set of log paths.

purge(**args*, ***kwargs*)
Purge old bans, jails and log files from database.

purgeage
Purge age in seconds.

updateDb(**args*, ***kwargs*)
Update an existing database, called during initialisation.

A timestamped backup is also created prior to attempting the update.

updateLog(**args*, ***kwargs*)
Updates hash and last position in log file.

Parameters **jail** : Jail

Jail of which the log file belongs to.

container : FileContainer

File container of the log file being updated.

fail2ban.server.database.**commitandrollback**(*f*)

4.2.6 fail2ban.server.datedetector module

class fail2ban.server.datedetector.**DateDetector**

Bases: object

Manages one or more date templates to find a date within a log line.

Attributes

templates	List of template instances managed by the detector.

Methods

addDefaultTemplate()	Add Fail2Ban's default set of date templates.
appendTemplate(template)	Add a date template to manage and use in search of dates.
getTime(line)	Attempts to return the date on a log line using templates.
matchTime(line)	Attempts to find date on a log line using templates.
sortTemplate()	Sort the date templates by number of hits

addDefaultTemplate()

Add Fail2Ban's default set of date templates.

appendTemplate(*template*)

Add a date template to manage and use in search of dates.

Parameters **template** : DateTemplate or str

Can be either a *DateTemplate* instance, or a string which will be used as the pattern for the *DatePatternRegex* template. The template will then be added to the detector.

Raises **ValueError**

If a template already exists with the same name.

getTime(*line*)

Attempts to return the date on a log line using templates.

This uses the templates' *getDate* method in an attempt to find a date.

Parameters **line** : str

Line which is searched by the date templates.

Returns float

The Unix timestamp returned from the first successfully matched template.

matchTime(*line*)

Attempts to find date on a log line using templates.

This uses the templates' *matchDate* method in an attempt to find a date. It also increments the match hit count for the winning template.

Parameters **line** : str

Line which is searched by the date templates.

Returns re.MatchObject

The regex match returned from the first successfully matched template.

sortTemplate()
Sort the date templates by number of hits

Sort the template lists using the hits score. This method is not called in this object and thus should be called from time to time. This ensures the most commonly matched templates are checked first, improving performance of matchTime and getTime.

templates
List of template instances managed by the detector.

4.2.7 fail2ban.server.datetemplate module

class fail2ban.server.datetemplate.**DateEpoch**
Bases: fail2ban.server.datetemplate.DateTemplate

A date template which searches for Unix timestamps.

This includes Unix timestamps which appear at start of a line, optionally within square braces (nsd), or on SELinux audit log lines.

Attributes

name	Name assigned to template.
regex	Regex used to search for date.

Methods

getDate(line)	Method to return the date for a log line.
getRegex()	
matchDate(line)	Check if regex for date matches on a log line.
setRegex(regex[, wordBegin])	Sets regex to use for searching for date in log line.

getDate(*line*)
Method to return the date for a log line.

Parameters **line** : str

Log line, of which the date should be extracted from.

Returns (float, str)

Tuple containing a Unix timestamp, and the string of the date which was matched and in turned used to calculated the timestamp.

getRegex()

matchDate(*line*)
Check if regex for date matches on a log line.

name
Name assigned to template.

regex
Regex used to search for date.

setRegex(*regex*, *wordBegin=True*)
Sets regex to use for searching for date in log line.

Parameters **regex** : str

The regex the template will use for searching for a date.

wordBegin : bool

Defines whether the regex should be modified to search at beginning of a word, by adding "b" to start of regex. Default True.

Raises **re.error**

If regular expression fails to compile

class fail2ban.server.datetemplate.**DatePatternRegex**(*pattern=None*)
Bases: fail2ban.server.datetemplate.DateTemplate

Date template, with regex/pattern

Parameters **pattern** : str

Sets the date templates pattern.

Attributes

name	Name assigned to template.
regex	Regex used to search for date.
pattern	The pattern used for regex with strptime "%" time fields.

Methods

getDate(line)	Method to return the date for a log line.
getRegex()	
matchDate(line)	Check if regex for date matches on a log line.
setRegex(value)	

getDate(*line*)
Method to return the date for a log line.

This uses a custom version of strptime, using the named groups from the instances *pattern* property.

Parameters **line** : str

Log line, of which the date should be extracted from.

Returns (float, str)

Tuple containing a Unix timestamp, and the string of the date which was matched and

in turned used to calculated the timestamp.

getRegex()

matchDate(*line*)
Check if regex for date matches on a log line.

name
Name assigned to template.

pattern
The pattern used for regex with strptime "%" time fields.

This should be a valid regular expression, of which matching string will be extracted from the log line. strptime style "%" fields will be replaced by appropriate regular expressions, or custom regex groups with names as per the strptime fields can also be used instead.

regex
Regex used to search for date.

setRegex(*value*)

class fail2ban.server.datetemplate.**DateTai64n**
Bases: fail2ban.server.datetemplate.DateTemplate

A date template which matches TAI64N formate timestamps.

Attributes

name	Name assigned to template.
regex	Regex used to search for date.

Methods

getDate(line)	Method to return the date for a log line.
getRegex()	
matchDate(line)	Check if regex for date matches on a log line.
setRegex(regex[, wordBegin])	Sets regex to use for searching for date in log line.

getDate(*line*)
Method to return the date for a log line.

Parameters **line** : str

Log line, of which the date should be extracted from.

Returns (float, str)

Tuple containing a Unix timestamp, and the string of the date which was matched and in turned used to calculated the timestamp.

getRegex()

matchDate(*line*)
Check if regex for date matches on a log line.

name
Name assigned to template.

regex
Regex used to search for date.

setRegex(*regex*, *wordBegin=True*)
Sets regex to use for searching for date in log line.

Parameters **regex** : str

The regex the template will use for searching for a date.

wordBegin : bool

Defines whether the regex should be modified to search at beginning of a word, by adding "b" to start of regex. Default True.

Raises **re.error**

If regular expression fails to compile

class fail2ban.server.datetemplate.**DateTemplate**
Bases: object

A template which searches for and returns a date from a log line.

This is an not functional abstract class which other templates should inherit from.

Attributes

name	Name assigned to template.
regex	Regex used to search for date.

Methods

getDate(line)	Abstract method, which should return the date for a log line
getRegex()	
matchDate(line)	Check if regex for date matches on a log line.
setRegex(regex[, wordBegin])	Sets regex to use for searching for date in log line.

getDate(*line*)
Abstract method, which should return the date for a log line

This should return the date for a log line, typically taking the date from the part of the line which matched the templates regex. This requires abstraction, therefore just raises exception.

Parameters **line** : str

Log line, of which the date should be extracted from.

Raises **NotImplementedError**

Abstract method, therefore always returns this.

getRegex()

matchDate(*line*)
Check if regex for date matches on a log line.

name
Name assigned to template.

regex
Regex used to search for date.

setRegex(*regex*, *wordBegin=True*)
Sets regex to use for searching for date in log line.

Parameters **regex** : str

The regex the template will use for searching for a date.

wordBegin : bool

Defines whether the regex should be modified to search at beginning of a word, by adding "b" to start of regex. Default True.

Raises **re.error**

If regular expression fails to compile

4.2.8 fail2ban.server.faildata module

class fail2ban.server.faildata.**FailData**

Methods

getLastReset()	
getLastTime()	
getMatches()	
getRetry()	
inc([matches])	
setLastReset(value)	
setLastTime(value)	
setRetry(value)	

getLastReset()

getLastTime()

getMatches()

getRetry()

inc(*matches=None*)

setLastReset(*value*)

setLastTime(*value*)

setRetry(*value*)

4.2.9 fail2ban.server.failmanager module

class fail2ban.server.failmanager.**FailManager**

Methods

	addFailure(ticket)
	cleanup(time)
	getFailTotal()
	getMaxRetry()
	getMaxTime()
	setFailTotal(value)
	setMaxRetry(value)
	setMaxTime(value)
	size()
	toBan()

addFailure(*ticket*)

cleanup(*time*)

getFailTotal()

getMaxRetry()

getMaxTime()

setFailTotal(*value*)

setMaxRetry(*value*)

setMaxTime(*value*)

size()

toBan()

exception fail2ban.server.failmanager.**FailManagerEmpty**
Bases: exceptions.Exception

args

message

4.2.10 fail2ban.server.failregex module

class fail2ban.server.failregex.**FailRegex**(*regex*)
Bases: fail2ban.server.failregex.Regex

Methods

	getHost()
	getMatchedLines()
	getMatchedTupleLines()
	getRegex()
	getSkippedLines()
	getUnmatchedLines()
	getUnmatchedTupleLines()
	hasMatched()
	search(tupleLines)

getHost()

getMatchedLines()

getMatchedTupleLines()

getRegex()

getSkippedLines()

getUnmatchedLines()

getUnmatchedTupleLines()

hasMatched()

search(*tupleLines*)

class fail2ban.server.failregex.**Regex**(*regex*)

Methods

	getMatchedLines()
	getMatchedTupleLines()
	getRegex()
	getSkippedLines()
	getUnmatchedLines()
	getUnmatchedTupleLines()
	hasMatched()
	search(tupleLines)

getMatchedLines()

getMatchedTupleLines()

getRegex()

getSkippedLines()

getUnmatchedLines()

getUnmatchedTupleLines()

hasMatched()

search(*tupleLines*)

exception fail2ban.server.failregex.**RegexException**

Bases: exceptions.Exception

args

message

4.2.11 fail2ban.server.filter module

class fail2ban.server.filter.**DNSUtils**

Methods

addr2bin(string)	Convert a string IPv4 address into an unsigned integer.
bin2addr(addr)	Convert a numeric IPv4 address into string n.n.n.n form.
cidr(i, n)	Convert an IP address string with a CIDR mask into a 32-bit integer.
dnsToIp(dns)	Convert a DNS into an IP address using the Python socket module.
isValidIP(string)	Return true if str is a valid IP
searchIP(text)	Search if an IP address if directly available and return it.
textToIp(text, useDns)	Return the IP of DNS found in a given text.

IP_CRE = <_sre.SRE_Pattern object at 0x7f42fed1c7b0>

static addr2bin(*string*)
Convert a string IPv4 address into an unsigned integer.

static bin2addr(*addr*)
Convert a numeric IPv4 address into string n.n.n.n form.

static cidr(*i*, *n*)
Convert an IP address string with a CIDR mask into a 32-bit integer.

static dnsToIp(*dns*)
Convert a DNS into an IP address using the Python socket module. Thanks to Kevin Drapel.

static isValidIP(*string*)
Return true if str is a valid IP

static searchIP(*text*)
Search if an IP address if directly available and return it.

static textToIp(*text*, *useDns*)
Return the IP of DNS found in a given text.

class fail2ban.server.filter.**FileContainer**(*filename*, *encoding*, *tail=False*)

Methods

close()	
getEncoding()	
getFileName()	
getHash()	
getPos()	
open()	
readline()	
setEncoding(encoding)	
setPos(value)	

close()

getEncoding()

getFileName()

getHash()

getPos()

open()

readline()

setEncoding(*encoding*)

setPos(*value*)

class fail2ban.server.filter.**FileFilter**(*jail*, ***kwargs*)
Bases: fail2ban.server.filter.Filter

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
status	Status of Filter plus files being monitored.

Methods

addBannedIP(ip)	
addFailRegex(value)	
addIgnoreIP(ip)	
addIgnoreRegex(value)	
addLogPath(path[, tail])	
containsLogPath(path)	
delFailRegex(index)	
delIgnoreIP(ip)	
delIgnoreRegex(index)	
delLogPath(path)	
findFailure(tupleLine[, date, ...])	
getDatePattern()	
getFailRegex()	
getFailures(filename)	
getFileContainer(path)	
getFindTime()	
getIgnoreCommand()	
getIgnoreIP()	
getIgnoreRegex()	
getLogEncoding()	
getLogPath()	
getMaxLines()	
getMaxRetry()	
getName()	
getUseDns()	
ignoreLine(tupleLines)	
inIgnoreIPList(ip)	
isAlive()	Return whether the thread is alive.
isDaemon()	
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
processLine(line[, date, returnRawHost, ...])	Split the time portion from log msg and return findFailures on them
processLineAndAdd(line[, date])	Processes the line for failures and populates failManager

Continued on next page

Table 4.42 – continued from previous page

run()	
setDaemon(daemonic)	
setDatePattern(pattern)	
setFindTime(value)	
setIgnoreCommand(command)	
setLogEncoding(encoding)	
setMaxLines(value)	
setMaxRetry(value)	
setName(name)	
setUseDns(value)	
start()	Sets active flag and starts thread.
stop()	Sets *active* property to False, to flag run method to return.

addBannedIP(*ip*)

addFailRegex(*value*)

addIgnoreIP(*ip*)

addIgnoreRegex(*value*)

addLogPath(*path*, *tail=False*)

containsLogPath(*path*)

daemon

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

delFailRegex(*index*)

delIgnoreIP(*ip*)

delIgnoreRegex(*index*)

delLogPath(*path*)

findFailure(*tupleLine*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)

getDatePattern()

getFailRegex()

getFailures(*filename*)

getFileContainer(*path*)

getFindTime()

getIgnoreCommand()

getIgnoreIP()

getIgnoreRegex()

getLogEncoding()

getLogPath()

getMaxLines()

getMaxRetry()

getName()

getUseDns()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

ignoreLine(*tupleLines*)

inIgnoreIPList(*ip*)

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

processLine(*line*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)
Split the time portion from log msg and return findFailures on them

processLineAndAdd(*line*, *date=None*)
Processes the line for failures and populates failManager

run()

setDaemon(*daemonic*)

setDatePattern(*pattern*)

setFindTime(*value*)

setIgnoreCommand(*command*)

setLogEncoding(*encoding*)

setMaxLines(*value*)

setMaxRetry(*value*)

setName(*name*)

setUseDns(*value*)

start()
Sets active flag and starts thread.

status
Status of Filter plus files being monitored.

stop()
Sets *active* property to False, to flag run method to return.

class fail2ban.server.filter.**Filter**(*jail*, *useDns='warn'*)
Bases: fail2ban.server.jailthread.JailThread

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
status	Status of failures detected by filter.

Methods

addBannedIP(ip)
addFailRegex(value)
addIgnoreIP(ip)
addIgnoreRegex(value)
delFailRegex(index)
delIgnoreIP(ip)
delIgnoreRegex(index)
findFailure(tupleLine[, date, ...])
getDatePattern()
getFailRegex()
getFindTime()
getIgnoreCommand()
getIgnoreIP()
getIgnoreRegex()
getMaxLines()
getMaxRetry()
getName()
getUseDns()

Continued on next page

Table 4.44 – continued from previous page

ignoreLine(tupleLines)	
inIgnoreIPList(ip)	
isAlive()	Return whether the thread is alive.
isDaemon()	
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
processLine(line[, date, returnRawHost, ...])	Split the time portion from log msg and return findFailures on them
processLineAndAdd(line[, date])	Processes the line for failures and populates failManager
run()	
setDaemon(daemonic)	
setDatePattern(pattern)	
setFindTime(value)	
setIgnoreCommand(command)	
setMaxLines(value)	
setMaxRetry(value)	
setName(name)	
setUseDns(value)	
start()	Sets active flag and starts thread.
stop()	Sets *active* property to False, to flag run method to return.

addBannedIP(*ip*)

addFailRegex(*value*)

addIgnoreIP(*ip*)

addIgnoreRegex(*value*)

daemon

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

delFailRegex(*index*)

delIgnoreIP(*ip*)

delIgnoreRegex(*index*)

findFailure(*tupleLine*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)

getDatePattern()

getFailRegex()

getFindTime()

getIgnoreCommand()

getIgnoreIP()

getIgnoreRegex()

getMaxLines()

getMaxRetry()

getName()

getUseDns()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

ignoreLine(*tupleLines*)

inIgnoreIPList(*ip*)

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

processLine(*line*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)
Split the time portion from log msg and return findFailures on them

processLineAndAdd(*line*, *date=None*)
Processes the line for failures and populates failManager

run()

setDaemon(*daemonic*)

setDatePattern(*pattern*)

setFindTime(*value*)

setIgnoreCommand(*command*)

setMaxLines(*value*)

setMaxRetry(*value*)

setName(*name*)

setUseDns(*value*)

start()
Sets active flag and starts thread.

status
Status of failures detected by filter.

stop()
Sets *active* property to False, to flag run method to return.

class fail2ban.server.filter.**JournalFilter**(*jail*, *useDns='warn'*)
Bases: fail2ban.server.filter.Filter

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
status	Status of failures detected by filter.

Methods

addBannedIP(ip)	
addFailRegex(value)	
addIgnoreIP(ip)	
addIgnoreRegex(value)	
addJournalMatch(match)	
delFailRegex(index)	
delIgnoreIP(ip)	
delIgnoreRegex(index)	
delJournalMatch(match)	
findFailure(tupleLine[, date, ...])	
getDatePattern()	
getFailRegex()	
getFindTime()	
getIgnoreCommand()	
getIgnoreIP()	
getIgnoreRegex()	
getJournalMatch(match)	
getMaxLines()	
getMaxRetry()	
getName()	
getUseDns()	
ignoreLine(tupleLines)	
inIgnoreIPList(ip)	
isAlive()	Return whether the thread is alive.

Continued on next page

Table 4.46 – continued from previous page

isDaemon()	
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
processLine(line[, date, returnRawHost, ...])	Split the time portion from log msg and return findFailures on them
processLineAndAdd(line[, date])	Processes the line for failures and populates failManager
run()	
setDaemon(daemonic)	
setDatePattern(pattern)	
setFindTime(value)	
setIgnoreCommand(command)	
setMaxLines(value)	
setMaxRetry(value)	
setName(name)	
setUseDns(value)	
start()	Sets active flag and starts thread.
stop()	Sets *active* property to False, to flag run method to return.

addBannedIP(*ip*)

addFailRegex(*value*)

addIgnoreIP(*ip*)

addIgnoreRegex(*value*)

addJournalMatch(*match*)

daemon

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

delFailRegex(*index*)

delIgnoreIP(*ip*)

delIgnoreRegex(*index*)

delJournalMatch(*match*)

findFailure(*tupleLine*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)

getDatePattern()

getFailRegex()

getFindTime()

getIgnoreCommand()

getIgnoreIP()

getIgnoreRegex()

getJournalMatch(*match*)

getMaxLines()

getMaxRetry()

getName()

getUseDns()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

ignoreLine(*tupleLines*)

inIgnoreIPList(*ip*)

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

processLine(*line*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)
Split the time portion from log msg and return findFailures on them

processLineAndAdd(*line*, *date=None*)
Processes the line for failures and populates failManager

run()

setDaemon(*daemonic*)

setDatePattern(*pattern*)

setFindTime(*value*)

setIgnoreCommand(*command*)

setMaxLines(*value*)

setMaxRetry(*value*)

setName(*name*)

setUseDns(*value*)

start()
Sets active flag and starts thread.

status
Status of failures detected by filter.

stop()
Sets *active* property to False, to flag run method to return.

4.2.12 fail2ban.server.filtergamin module

4.2.13 fail2ban.server.filterpoll module

class fail2ban.server.filterpoll.**FilterPoll**(*jail*)
Bases: fail2ban.server.filter.FileFilter

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
status	Status of Filter plus files being monitored.

Methods

addBannedIP(ip)
addFailRegex(value)
addIgnoreIP(ip)
addIgnoreRegex(value)
addLogPath(path[, tail])
containsLogPath(path)
delFailRegex(index)
delIgnoreIP(ip)
delIgnoreRegex(index)
delLogPath(path)
findFailure(tupleLine[, date, ...])
getDatePattern()
getFailRegex()
getFailures(filename)
getFileContainer(path)
getFindTime()
getIgnoreCommand()

Continued on next page

Table 4.48 – continued from previous page

getIgnoreIP()	
getIgnoreRegex()	
getLogEncoding()	
getLogPath()	
getMaxLines()	
getMaxRetry()	
getName()	
getUseDns()	
ignoreLine(tupleLines)	
inIgnoreIPList(ip)	
isAlive()	Return whether the thread is alive.
isDaemon()	
isModified(filename)	
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
processLine(line[, date, returnRawHost, ...])	Split the time portion from log msg and return findFailures on them
processLineAndAdd(line[, date])	Processes the line for failures and populates failManager
run()	
setDaemon(daemonic)	
setDatePattern(pattern)	
setFindTime(value)	
setIgnoreCommand(command)	
setLogEncoding(encoding)	
setMaxLines(value)	
setMaxRetry(value)	
setName(name)	
setUseDns(value)	
start()	Sets active flag and starts thread.
stop()	Sets *active* property to False, to flag run method to return.

addBannedIP(*ip*)

addFailRegex(*value*)

addIgnoreIP(*ip*)

addIgnoreRegex(*value*)

addLogPath(*path*, *tail=False*)

containsLogPath(*path*)

daemon

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

delFailRegex(*index*)

delIgnoreIP(*ip*)

delIgnoreRegex(*index*)

delLogPath(*path*)

findFailure(*tupleLine*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)

getDatePattern()

getFailRegex()

getFailures(*filename*)

getFileContainer(*path*)

getFindTime()

getIgnoreCommand()

getIgnoreIP()

getIgnoreRegex()

getLogEncoding()

getLogPath()

getMaxLines()

getMaxRetry()

getName()

getUseDns()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

ignoreLine(*tupleLines*)

inIgnoreIPList(*ip*)

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

isModified(*filename*)

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

processLine(*line*, *date=None*, *returnRawHost=False*, *checkAllRegex=False*)
Split the time portion from log msg and return findFailures on them

processLineAndAdd(*line*, *date=None*)
Processes the line for failures and populates failManager

run()

setDaemon(*daemonic*)

setDatePattern(*pattern*)

setFindTime(*value*)

setIgnoreCommand(*command*)

setLogEncoding(*encoding*)

setMaxLines(*value*)

setMaxRetry(*value*)

setName(*name*)

setUseDns(*value*)

start()
Sets active flag and starts thread.

status
Status of Filter plus files being monitored.

stop()
Sets *active* property to False, to flag run method to return.

4.2.14 fail2ban.server.filterpyinotify module

4.2.15 fail2ban.server.filtersystemd module

4.2.16 fail2ban.server.jail module

class fail2ban.server.jail.**Jail**(*name*, *backend='auto'*, *db=None*)
Fail2Ban jail, which manages a filter and associated actions.

The class handles the initialisation of a filter, and actions. It's role is then to act as an interface between the filter and actions, passing bans detected by the filter, for the actions to then act upon.

Parameters **name** : str

Name assigned to the jail.

backend : str

Backend to be used for filter. "auto" will attempt to pick the most preferred backend method. Default: "auto"

db : Fail2BanDb

Fail2Ban persistent database instance. Default: *None*

Attributes

name	Name of jail.
database	The database used to store persistent data for the jail.
filter	The filter which the jail is using to monitor log files.
actions	Actions object used to manage actions for jail.
idle	A boolean indicating whether jail is idle.
status	The status of the jail.

Methods

getFailTicket()	Get a fail ticket from the jail.
is_alive()	Check jail "is_alive" by checking filter and actions threads.
putFailTicket(ticket)	Add a fail ticket to the jail.
start()	Start the jail, by starting filter and actions threads.
stop()	Stop the jail, by stopping filter and actions threads.

actions
Actions object used to manage actions for jail.

database
The database used to store persistent data for the jail.

filter
The filter which the jail is using to monitor log files.

getFailTicket()
Get a fail ticket from the jail.

Used by actions to get a failure for banning.

idle
A boolean indicating whether jail is idle.

is_alive()
Check jail "is_alive" by checking filter and actions threads.

name
Name of jail.

putFailTicket(*ticket*)
Add a fail ticket to the jail.

Used by filter to add a failure for banning.

start()
Start the jail, by starting filter and actions threads.

Once stated, also queries the persistent database to reinstate any valid bans.

status
The status of the jail.

stop()
Stop the jail, by stopping filter and actions threads.

4.2.17 fail2ban.server.jails module

class fail2ban.server.jails.**Jails**
Bases: _abcoll.Mapping

Handles the jails.

This class handles the jails. Creation, deletion or access to a jail must be done through this class. This class is thread-safe which is not the case of the jail itself, including filter and actions. This class is based on Mapping type, and the *add* method must be used to add additional jails.

Methods

add(name, backend[, db])	Adds a jail.
get((k[,d]) -> D[k] if k in D, ...)	
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
values(() -> list of D's values)	

add(*name*, *backend*, *db=None*)
Adds a jail.

Adds a new jail if not already present which should use the given backend.

Parameters **name** : str

The name of the jail.

backend : str

The backend to use.

db : Fail2BanDb

Fail2Ban's persistent database instance.

Raises **DuplicateJailException**

If jail name is already present.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

values() → list of D's values

4.2.18 fail2ban.server.jailthread module

class fail2ban.server.jailthread.**JailThread**
Bases: threading.Thread

Abstract class for threading elements in Fail2Ban.

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.
status	Abstract - Should provide status information.

active	(bool) Control the state of the thread.
idle	(bool) Control the idle state of the thread.
sleeptime	(int) The time the thread sleeps for in the loop.

Methods

getName()	
isAlive()	Return whether the thread is alive.
isDaemon()	
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
run()	Abstract - Called when thread starts, thread stops when returns.
setDaemon(daemonic)	
setName(name)	
start()	Sets active flag and starts thread.
stop()	Sets *active* property to False, to flag run method to return.

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

getName()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

run()
Abstract - Called when thread starts, thread stops when returns.

setDaemon(*daemonic*)

setName(*name*)

start()
Sets active flag and starts thread.

status
Abstract - Should provide status information.

stop()
Sets *active* property to False, to flag run method to return.

4.2.19 fail2ban.server.mytime module

class fail2ban.server.mytime.**MyTime**

Attributes

myTime	

Methods

	gmtime()
	localtime([x])
	now()
	setTime(t)
	time()

static gmtime()

static localtime(*x=None*)

myTime = None

static now()

static setTime(*t*)

static time()

4.2.20 fail2ban.server.server module

class fail2ban.server.server.**Server**(*daemon=False*)

Methods

	addAction(name, value, *args)
	addFailRegex(name, value)
	addIgnoreIP(name, ip)
	addIgnoreRegex(name, value)
	addJail(name, backend)
	addJournalMatch(name, match)
	addLogPath(name, fileName[, tail])
	delAction(name, value)
	delFailRegex(name, index)
	delIgnoreIP(name, ip)
	delIgnoreRegex(name, index)
	delJail(name)
	delJournalMatch(name, match)
	delLogPath(name, fileName)
	flushLogs()
	getAction(name, value)
	getActions(name)
	getBanTime(name)
	getDatabase()
	getDatePattern(name)
	getFailRegex(name)
	getFindTime(name)
	getIdleJail(name)
	getIgnoreCommand(name)
	getIgnoreIP(name)

Continued on next page

Table 4.55 – continued from previous page

getIgnoreRegex(name)
getJournalMatch(name)
getLogEncoding(name)
getLogLevel()
getLogPath(name)
getLogTarget()
getMaxLines(name)
getMaxRetry(name)
getUseDns(name)
quit()
setBanIP(name, value)
setBanTime(name, value)
setDatabase(filename)
setDatePattern(name, pattern)
setFindTime(name, value)
setIdleJail(name, value)
setIgnoreCommand(name, value)
setLogEncoding(name, encoding)
setLogLevel(value)
setLogTarget(target)
setMaxLines(name, value)
setMaxRetry(name, value)
setUnbanIP(name, value)
setUseDns(name, value)
start(sock, pidfile[, force])
startJail(name)
status()
statusJail(name)
stopAllJail()
stopJail(name)

addAction (*name*, *value*, **args*)

addFailRegex (*name*, *value*)

addIgnoreIP (*name*, *ip*)

addIgnoreRegex (*name*, *value*)

addJail (*name*, *backend*)

addJournalMatch (*name*, *match*)

addLogPath (*name*, *fileName*, *tail=False*)

delAction (*name*, *value*)

delFailRegex (*name*, *index*)

delIgnoreIP (*name*, *ip*)

delIgnoreRegex (*name*, *index*)

delJail (*name*)

delJournalMatch (*name*, *match*)

delLogPath (*name*, *fileName*)

flushLogs()

getAction(*name*, *value*)

getActions(*name*)

getBanTime(*name*)

getDatabase()

getDatePattern(*name*)

getFailRegex(*name*)

getFindTime(*name*)

getIdleJail(*name*)

getIgnoreCommand(*name*)

getIgnoreIP(*name*)

getIgnoreRegex(*name*)

getJournalMatch(*name*)

getLogEncoding(*name*)

getLogLevel()

getLogPath(*name*)

getLogTarget()

getMaxLines(*name*)

getMaxRetry(*name*)

getUseDns(*name*)

quit()

setBanIP(*name*, *value*)

setBanTime(*name*, *value*)

setDatabase(*filename*)

setDatePattern(*name*, *pattern*)

setFindTime(*name*, *value*)

setIdleJail(*name*, *value*)

setIgnoreCommand(*name*, *value*)

setLogEncoding(*name*, *encoding*)

setLogLevel(*value*)

setLogTarget(*target*)

setMaxLines(*name*, *value*)

setMaxRetry(*name*, *value*)

setUnbanIP(*name*, *value*)

setUseDns(*name*, *value*)

start(*sock*, *pidfile*, *force=False*)

startJail(*name*)

status()

statusJail(*name*)

stopAllJail()

stopJail(*name*)

exception fail2ban.server.server.**ServerInitializationError**
Bases: exceptions.Exception

args

message

4.2.21 fail2ban.server.strptime module

fail2ban.server.strptime.**reGroupDictStrptime**(*found_dict*)
Return time from dictionary of strptime fields

This is tweaked from python built-in _strptime.

Parameters **found_dict** : dict

Dictionary where keys represent the strptime fields, and values the respective value.

Returns float

Unix time stamp.

4.2.22 fail2ban.server.ticket module

class fail2ban.server.ticket.**BanTicket**(*ip*, *time*, *matches=None*)
Bases: fail2ban.server.ticket.Ticket

Methods

getAttempt()	
getIP()	
getMatches()	
getTime()	
setAttempt(value)	
setIP(value)	
setTime(value)	

getAttempt()

getIP()

getMatches()

getTime()

setAttempt(*value*)

setIP(*value*)

setTime(*value*)

class fail2ban.server.ticket.**FailTicket**(*ip*, *time*, *matches=None*)
Bases: fail2ban.server.ticket.Ticket

Methods

getAttempt()
getIP()
getMatches()
getTime()
setAttempt(value)
setIP(value)
setTime(value)

getAttempt()

getIP()

getMatches()

getTime()

setAttempt(*value*)

setIP(*value*)

setTime(*value*)

class fail2ban.server.ticket.**Ticket**(*ip*, *time*, *matches=None*)

Methods

getAttempt()
getIP()
getMatches()
getTime()
setAttempt(value)
setIP(value)
setTime(value)

getAttempt()

getIP()

getMatches()

getTime()

setAttempt(*value*)

setIP(*value*)

setTime(*value*)

4.2.23 fail2ban.server.transmitter module

class fail2ban.server.transmitter.**Transmitter**(*server*)

Methods

proceed(command)	
status(command)	

proceed(*command*)

status(*command*)

4.3 fail2ban.exceptions module

Fail2Ban exceptions used by both client and server

exception fail2ban.exceptions.**DuplicateJailException**
Bases: exceptions.Exception

args

message

exception fail2ban.exceptions.**UnknownJailException**
Bases: exceptions.KeyError

args

message

4.4 fail2ban.helpers module

class fail2ban.helpers.**FormatterWithTraceBack**(*fmt*, **args*, ***kwargs*)
Bases: logging.Formatter

Custom formatter which expands %(tb) and %(tbc) with tracebacks

TODO: might need locking in case of compressed tracebacks

Methods

converter(...)	Convert seconds since the Epoch to a time tuple expressing local time.
format(record)	
formatException(ei)	Format and return the specified exception information as a string.
formatTime(record[, datefmt])	Return the creation time of the specified LogRecord as formatted text.
usesTime()	Check if the format uses the creation time of the record.

converter(*[seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min, tm_sec,tm_wday,tm_yday,tm_isdst*)
Convert seconds since the Epoch to a time tuple expressing local time. When 'seconds' is not passed in,

convert the current time instead.

format(*record*)

formatException(*ei*)
Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

formatTime(*record*, *datefmt=None*)
Return the creation time of the specified LogRecord as formatted text.

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the 'converter' attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the 'converter' attribute in the Formatter class.

usesTime()
Check if the format uses the creation time of the record.

class fail2ban.helpers.**TraceBack**(*compress=False*)
Bases: object

Customized traceback to be included in debug messages

Methods

__call__()

fail2ban.helpers.**excepthook**(*exctype*, *value*, *traceback*)
Except hook used to log unhandled exceptions to Fail2Ban log

fail2ban.helpers.**formatExceptionInfo**()
Consistently format exception information

fail2ban.helpers.**getLogger**(*name*)
Get logging.Logger instance with Fail2Ban logger name convention

fail2ban.helpers.**mbasename**(*s*)
Custom function to include directory name if filename is too common

Also strip .py at the end

4.5 fail2ban.protocol module

fail2ban.protocol.**printFormatted**()

fail2ban.protocol.**printWiki**()

4.6 fail2ban.version module

CHAPTER 5

Indices and tables

- *genindex*
- *search*

f