Fussy Documentation

Release 1.0.6

VRPlumber Consulting Inc.

June 16, 2012

CONTENTS

Fussy provide "embedded firmware" style upgrade mechanics (field upgrades) where a user uploads a (signed, possibly encrypted) package to your device and you want to verify, unpack and switch to running with that package. It provides a back-off mechanic where a "failsafe" software bundle can be installed to which the system will back off in the event of a firmware upgrade failure.

Fussy is focussed on Linux machines, and should not be expected to work on any other platform. It expects to work on fairly "full" distributions, such as an Ubuntu Server LTS platform, but even a fairly minimal Debian distribution should work.

If you are more interested in desktop auto-updaters, the Esky project is likely closer to your requirements.

CHAPTER

ONE

INSTALLATION

Standard installation (note: rsync is only required to install firmware, not pack it):

```
$ apt-get install tar rsync gnupg
$ virtualenv fussy-env
$ source fussy-env/bin/activate
$ pip install fussy
```

If you would like to extend/enhance Fussy, see *Contributing to Fussy*

CHAPTER

TWO

FIRMWARE STRUCTURE

Your Fussy-packaged firmware needs to be "self contained" in that all of your upgradable code resides within a single directory, with any system-level hooks added by symlinking from required directory into your package.

A minimal package named "my-package" might look like this on disk:

```
/opt/my-package/current -> firmware-2012-04-12T13.30.03.796749
/opt/my-package/firmware-2012-04-12T13.30.03.796749
/opt/my-package/firmware-2012-04-12T13.30.03.796749/.pre-install
/opt/my-package/firmware-2012-04-12T13.30.03.796749/.post-install
/opt/my-package/firmware-2012-04-12T13.30.03.796749/etc/fussy/keys
```

and the system would contain links such that various components of the firmware would appear in the "correct" spot on the filesystem:

```
/etc/fussy -> /opt/my-package/current/etc/fussy
```

So that, on the (atomic) switch between any two versions of the software (the updating of the *current* link), the new version of the software will be picked up by all processes.

CHAPTER

THREE

QUICK START

With the above structure, and *Encryption Setup*, the use of fussy looks like this (the -e argument says to encrypt for the given client key):

```
$ fussy-pack -r sample-setup -e C064B21C
...
Enter passphrase:
/tmp/fussy-bsPsty-pack/sample-setup.tar.gz.gpg
$ fussy-install /tmp/fussy-bsPsty-pack/sample-setup.tar.gz.gpg
...
$ la /opt/firmware/
total 24K
drwxr-xr-x 6 mcfletch root  4.0K 2012-04-12 12:01 .
drwxr-xr-x 4 root     root  4.0K 2012-04-12 11:14 ..
lrwxrwxrwx 1 mcfletch users   28 2012-04-12 12:01 current -> /opt/firmware/sample-setup-3
drwxr-xr-x 3 mcfletch users 4.0K 2012-04-12 11:25 sample-setup
drwxr-xr-x 3 mcfletch users 4.0K 2012-04-12 11:26 sample-setup-1
drwxr-xr-x 3 mcfletch users 4.0K 2012-04-12 11:37 sample-setup-2
drwxr-xr-x 3 mcfletch users 4.0K 2012-04-12 12:01 sample-setup-3
```

Obviously *fussy-install* would normally be run on a different machine than the machine doing *fussy-pack*. The machine doing the *fussy-install* needs to be properly set up with the structure described above.

CHAPTER

FOUR

CONTENTS

4.1 Encryption Setup

GnuPG is used for encryption:

- Create your own key, this is the key you will use to sign your distributions,

```
$ gpg --gen-key
...
$ gpg -K
...
sec   4096R/6EB7E943
...
$ gpg --export-secret-key -a > secret.key # BACK THIS UP!
...
```

- Back this key up. It will be the only key your machines will trust!
- You will need this key on your build machines to perform the encryption
- Create a keyring that will be installed on the embedded platform, this is, by default, located at */etc/fussy/keys*

```
$ mkdir -p /etc/fussy/keys
$ chmod 0700 /etc/fussy/keys
$ gpg --export -a 6EB7E943 | GNUPGHOME=/etc/fussy/keys gpg --import
$ GNUPGHOME=/etc/fussy/keys gpg --edit-key 6EB7E943 trust quit
# Mark as ultimately trusted...
```

- Optionally, generate a private key for the machines to allow for obfuscating the firmware with encryption

```
  $ GNUPGHOME=/etc/fussy/keys gpg --gen-key
  $ GNUPGHOME=/etc/fussy/keys gpg -K
  ...
  sec   4096R/C064B21C
  $ GNUPGHOME=/etc/fussy/keys gpg --export -a C064B21C | gpg --import

* Anyone with access to the machine can trivially extract the key *or* the
  unpacked content of your packages, do *not* rely on the encryption to
  prevent access to the package contents.
```

- Install the /etc/fussy/keys key-ring on your "embedded" servers (i.e. add the directory to your system-build process)

4.2 Packing Your Firmware

Your firmware will be packed into a *.tar.gz.gpg* format for delivery to your customers. The firmware *SHOULD* include the fussy package, as well as whatever mechanism you are using to deliver the new firmware packages to your devices. Your system *should* also make the */etc/fussy* directory point into your firmware package so that you can update the keys your machines trust.

4.2.1 Install Scripts

The two installation scripts *.pre-install* and *.post-install* are entry points which allow you to control your software before and after the installation process completes. Normally these scripts should shut down and restart services, perform pre-upgrade backups of data, migrate databases, etc. In some cases, they may need to e.g. create system-level symlinks, run apt updates or the like.

These scripts are critical failure points. They must be coded defensively and idempotently, they potentially can be run "backward" (that is, a downgrade rather than an upgrade), or update from extremely old firmware versions. If they fail to run properly, a firware installation failure will occur, and your machine will fall back to the failsafe firmware.

4.2.2 fussy-pack

fussy-pack packs a firmware image into a *.tar.gz.gpg* file suitable to be delivered to the machine (for example, by a user uploading it).

```
$ fussy-pack --help
Usage: fussy-pack [options]

Options:
  -h, --help            show this help message and exit
  -x EXCLUDE, --exclude=EXCLUDE
                        Paths/patterns to exclude from the archive
  -r ROOT, --root=ROOT  The root of the directory hierarchy (where tar will be
                        run)
  -e ENCRYPT, --encrypt-for=ENCRYPT
                        The name of the key for which to encrypt (otherwise
                        just sign)
```

Fussy packing should be integrated into your build script such that it runs on a directory which is uniquely named, for instance:

```
$ sudo fussy-pack -r build/firmware-2012-04-12T18.32.45
```

where the name *firmware-...* is the name of directory to appear in the final */opt/mypackage* directory.

```
$ mkdir firmware-2012-04-12T18.32.45
...
$ fussy-pack -r firmware-2012-04-12T18.32.45 --exclude .coverage
```

You will be prompted for your gpg pass-phrase during the packing phase if you set a password on your GPG key. The result will be a firmware package with a name such as:

```
/tmp/fussy-1341234-pack/firmware-2012-04-12T18.32.45.tar.gz.gpg
```

with the filename printed to stdout at the end of the build process.

4.2.3 Module: fussy.pack

Bundle a package as a signed firmware for installation/redistribution

`fussy.pack.get_options()`
Produces the OptionParser for :func:main

`fussy.pack.main()`
Main function for the packing script

`fussy.pack.pack(root_dir, excludes=None, encrypt_for=None)`
Bundle directory into a firmware file...

- **root_dir – directory to be packed into the firmware, the** `os.path.basename()` of this directory will be the name used in the created bundle filename and the directory installed into the target on client machines
- **excludes – patterns to exclude from the firmware image (passed to tar** during packing)
- **encrypt_for – if specified also encrypts for the given key, should** be the key ID, fingerprint, or (unique) email

returns absolute filename for generated gpg firmware (created in a temporary directory)

4.3 Installing Your Firmware

Your packed firmware needs to be uploaded to your system (fussy does not provide this functionality). Once uploaded, you need to arrange to call the fussy-install script. Keep in mind that this script likely needs to be called as root (in order to allow it to properly set ownership and start/restart services).

To install a package into the default */opt/firmware/* location:

```
$ fussy-install /tmp/fussy-iHUyhV-pack/sample-setup.tar.gz.gpg
$ fussy-clean
```

this will update the */opt/firmware/current* link to point to the unpacked version of the firmware and run the .pre-install and .post-install scripts with the final location of the firmware as their first argument.

If you install the same firmware twice, the firmware's directory name will be altered to have a trailing integer suffix.

If the installation fails, then the *previous* current installation will be re-installed (i.e. will be linked, then have its pre and post-install scripts run.

4.3.1 fussy-install

fussy-install unpacks the firmware into a temporary directory, rsyncs it into the firmware target directory, runs the *.pre-install* script, if present, then "fixates" the version by linking *current* to the new version, and finally runs the *.post-install* script, if present.

```
$ fussy-install --help
Usage: fussy-install [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  The firmware archive to unpack, must be a .tar.gz.gpg
                        or a .tar.gz.asc
  -k KEYRING, --keyring=KEYRING
                        GPG keyring to use for verification/decryption
```

```
                        (default /etc/fussy/keys)
  -t TARGET, --target=TARGET
                        Directory into which to rsync the firmware (default
                        /opt/firmware)
```

4.3.2 fussy-clean

```
$ fussy-clean --help
Usage: fussy-clean [options]

Options:
  -h, --help            show this help message and exit
  -t TARGET, --target=TARGET
                        Directory into which to rsync the firmware (default
                        /opt/firmware)
```

4.3.3 Module: fussy.install

Install a given firmware package onto the file system (commands fussy-install and fussy-clean)

Major TODO items:

- TODO: error handling for all of the big issues (disk-space, memory, script failures)

`fussy.install.`**`clean`**(*target='/opt/firmware'*, *protected=None*)
Naive cleaning implementation

Removes all names in target which are not in protected, paths in protected must be *full* path names, as returned by glob. The target of the current link is protected.

`fussy.install.`**`clean_main`**()
Main entry-point for fussy-clean script

Steps taken:

•parses arguments

•launches `clean()`

`fussy.install.`**`enable`**(*final_target*, *current*)
Attempt to enable final_target as the current release

Steps taken:

•runs *final_target/.pre-install final_target* (iff .pre-install is present)

•(atomically) swaps the link *current* for a link that points to *final_target*

•runs *final_target/.post-install final_target* (iff .post-install is present)

•if a failure occurs before swap-link completes, deletes final_target

returns None raises Exceptions on lots of failure cases

`fussy.install.`**`ensure_current_link`**(*current*, *failsafe*)
Ensure that current is a link (not a directory)

`fussy.install.`**`final_path`**(*link*)
Get the final path of the given link

raises IOError if link target does not exist, or the target is not a directory

returns normalized real target path of the link

fussy.install.**get_options**()
Creates the OptionParser used in `main()`

fussy.install.**install**(*filename*, *keyring='/etc/fussy/keys'*, *target='/opt/firmware'*)
Install given firmware <filename> into given target directory

Steps taken:

- unpack firmware (using `fussy.install.install_bytes()`)
- enable firmware (using `fussy.install.enable()`)
- if :func:enable fails, enable *previous* (or *failsafe* if there was no previous)

returns (error_code (0 is success), path name of the installed package)

fussy.install.**install_bytes**(*filename*, *keyring='/etc/fussy/keys'*, *target='/opt/firmware'*)
Install the packaged bytes into a final target directory

Steps taken:

- unpack firmware using `fussy.unpack.unpack()`
- **rsync new_firmware into /opt/firmware (*target*)**
  - if *CURRENT_LINK* (current) is present in *target*, will hard-link shared files between the new firmware and *current* to reduce disk use (using **rsync** parameter –link-dest)
- removes the temporary directory where unpacking was performed

returns full path to sub-directory of target where new firmware was installed

raises Errors on most failures, including disk-full, failed commands, missing executables, etc

fussy.install.**main**()
Main entry-point for the fussy-install script

Steps taken:

- parses arguments
- launches `install()`

fussy.install.**swap_link**(*final_target*, *current*)
Swap current link to point to final_target

Steps taken:

- if there is an existing tmp link, remove it
- create a tmp link to the final target
- rename tmp link to *current*

returns None

4.4 Contributing to Fussy

Fussy is a very young project, so feel free to pitch in if you have things you want to share. The code is hosted on Launchpad and managed using *bzr*. You can email the author as well.

Source code installation for contribution/extension:

```
$ apt-get install bzr tar rsync gnupg
$ virtualenv fussy-env
$ source fussy-env/bin/activate
$ bzr branch lp:fussy
$ cd fussy
$ python setup.py develop
$ pip install nose globalsub mockproc coverage
$ cd tests
$ ./covertest.sh
```

Feel free to issue pull requests in Launchpad if you have enhancements you feel you need.

4.5 Non-blocking IO Streams

The utility module fussy.nbio provides a basic mechanism to stream data through a series of processes.

The nbio module allows you to fairly easily create (potentially long) chains of processes which can all be run in parallel. The pipe internally uses non-blocking IO and generators to allow it to prevent buffer full deadlocks.

```
>>> from fussy.nbio import Process as P
>>> import requests
>>> response = requests.get( 'http://www.vrplumber.com' )
>>> pipe = response.iter_content | P( 'gzip -c' )
>>> result = pipe()
>>> assert result.startswith( '\x1f\x8b' )
```

Create a process and read output:

```
>>> from fussy.nbio import Process as P
>>> pipe = P( 'echo "Hello world"' )
>>> pipe()
'Hello world\n'
```

Pipe the output through another process:

```
>>> from fussy.nbio import Process as P
>>> pipe = P( 'echo "Hello world"' ) | P( 'cat' )
>>> pipe()
'Hello world\n'
```

Pipe a generator into a pipe:

```
>>> from fussy.nbio import Process as P
>>> def gen():
...     for i in range( 20 ):
...         yield str(i)
...
>>> pipe = gen() | P( 'cat' )
>>> pipe()
'01234567891011121314151617181 9'
```

Pipe a pipe into a function (note: the output also gets added to the pipe's output):

```
>>> from fussy.nbio import Process as P
>>> result = []
>>> pipe = P( 'echo "Hello world"' ) | result.append
>>> pipe()
'Hello world\n'
```

```
>>> result
['Hello world\n']
```

Pipe a pipe to our current stdout:

```
>>> from fussy.nbio import Process as P
>>> pipe = P( 'echo "Hello world"' ) | '-'
>>> pipe()
Hello world
''
```

4.5.1 Module: fussy.nbio

Wraps subprocess with pipe semantics and generator based multiplexing

pipe = open(somefile, 'rb') | nbio.Process(['grep','blue']) | nbio.Process(['wc','-l'])

exception `fussy.nbio.`**NBIOError**
Base class for nbio errors

__weakref__
list of weak references to the object (if defined)

class `fussy.nbio.`**Pipe**(**processes*)
Pipe of N processes which all need to process data in parallel

__call__(*pause_on_silence=0.01*)
Iterate over this pipeline, returning combined results as a string

__getitem__(*index*)
Retrieve a particular item in the pipe

__gt__(*other*)
Pipe our output into a file

__iter__()
Iterate over the processes in the pipe

If the stdout/stderr of the processes is not captured, then we will yield the results in whatever chunk-size is yielded from the individual processes.

If all of the processes yield DID_NOTHING in a particular cycle, then the pipe will do a pause() for self.pause_on_silence (normally passed into the __call__) before the next iteration.

__len__()
Return the number of items in this pipe

__lt__(*other*)
Pipe input from a named file

__or__(*other*)
Pipe our output into a process, callable or list

__ror__(*other*)
Pipe output of other into our first item

__weakref__
list of weak references to the object (if defined)

append(*process*)
Add the given PipeComponent to this pipe (note: does not connect stdin/stdout)

first
Retrieves the first item in the pipe

get_component(*other*)
Given a python object other, create a PipeComponent for it

The purpose of this method is to allow for fairly "natural" descriptions of tasks. You can pipe to or from files, to or from the string '-' (stdin/stdout), to the string '' (collect stdout), or from a regular string (which is treated as input). You can pipe iterables into a pipe, you can pipe the result of pipes into callables.

last
Retrieves the last item in the pipe

prepend(*process*)
Add the given PipeComponent to this pipe (note: does not connect stdin/stdout)

class fussy.nbio.**Process**(*command*, *stderr=False*, *stdout=True*, *stdin=True*, ***named*)
A particular process in a Pipe

Processes are the most common entry point when using nbio, you create processes and pipe data into or out of them as appropriate to create Pipes.

Under the covers the Process runs subprocess.Popen, and it accepts most of the fields subprocess.Popen does. By default it captures stdout and pipes data into stdin. If nothing is connected to stdin then stdin is closed on the first iteration of the pipe. If nothing is connected to stdout or stderr (if stderr is captured) then the results will be returned to the caller joined together with ''

The implication is that if you do not want to store all of the results in RAM, you need to "sink" the results into a process or file, or *not* capture the results (pass False for stdout or stderr).

__call__(**args*, ***named*)
Create a Pipe and run it with just this item as its children

__gt__(*other*)
Pipe our output into a filename

__init__(*command*, *stderr=False*, *stdout=True*, *stdin=True*, ***named*)
Initialize the Process

command – subprocess.Popen command string or list if a string, and "shell" is not explicitly set, then will set "shell=True"

stdin – whether to provide stdin writing

stdout – whether to capture stdout

stderr – whether to capture stderr, if -1, then combine stdout and stderr

good_exit – if provided, iterable which provides the set of good exit codes which will not raise ProcessError when encountered

by_line – if provided, will cause the output to be line-buffered so that only full lines will be reported, the 'n' character will be used to split the output, so there will be no 'n' character at the end of each line.

named – passed to the subprocess.Popen() command

__iter__()
Iterate over the results of the process (normally done by the Pipe)

__lt__(*other*)
Pipe our input from a filename

__or__(*other*)
Pipe our output into a process, callable or list

pipe = Pipe(Process('cat test.txt') | Process('grep blue') | []) pipe()

__ror__(*other*)
Pipe other into self

check_exit()
Check our exit code

iter_read()
Create the thing which iterates our read operation

iter_write(*source*)
Create a thing which will read from source and write to us

kill()
Kill our underlying subprocess.Popen

start_pipe(*stdin*, *stdout*, *stderr*, ***named*)
Start the captive process (internal operation)

exception fussy.nbio.**ProcessError**
Called process returned an error code

Attributes:

process – the Process (if applicable) which raised the error

fussy.nbio.**by_line**(*iterable*)
Buffer iterable yielding individual lines

fussy.nbio.**close**(*fh*)
Close the file/socket/closable thing

fussy.nbio.**fileno**(*fh*)
Determine the fileno for the file-like thing

fussy.nbio.**pause**(*duration*)
Allow tests to override sleeping using globalsub

fussy.nbio.**reader**(*fh*, *blocksize=4096*)
Produce content blocks from fh without blocking

fussy.nbio.**writeiter**(*iterator*, *fh*)
Write content from iterator to fh

To write a file from a read file:

```
writeiter(
    reader( open( filename )),
    fh
)
```

To write a request.response object into a tar pipe iteratively:

```
writeiter(
    response.iter_content( 4096, decode_unicode=False ),
    pipe
)
```

fussy.nbio.**writer**(*content*, *fh*, *encoding='utf8'*)
Continue writing content (string) to fh until content is consumed

Used by writeiter to writing individual bits of content to the fh

4.6 Tree Linking

This utility module provides a function to create a tree of links which point from your system's standard directories into your fussy distribution directories.

```
$ fussy-link-tree /opt/firmware/current/etc /etc
```

4.6.1 Module: fussy.linktree

Link all files in a distribution from relative location on the hard disk

`fussy.linktree.`**`get_options`**`()`
Creates the OptionParser used in `main()`

`fussy.linktree.`**`link_tree`**(*distribution_dir*, *target_dir='/'*, *symbolic=True*)
(Sym)link all files under distribution_dir from target_dir

4.7 Cron Lock

The `fussy.cronlock` module provides a simple mechanism to prevent runaway cron jobs. A lock can be used to protect a piece of code that should only ever have a single running instance:

```
from fussy import cronlock
lock = cronlock.Lock( 'test.lock' )

with lock:
    do_something()
```

It can also be used to prevent a piece of code from running for too long (note: the lock does not need to be held for this to work):

```
from fussy import cronlock
lock = cronlock.Lock( 'test.lock' )

lock.set_timeout( 20 ) # from this moment
do_something_that_might_take_a_while()
```

A decorator is provided to create and hold the lock for a given function, normally the main function of your cron-lock:

```
from fussy import with_lock

@with_lock( 'test.lock', timeout=20 )
def main():
    """Your cron job main-loop"""
```

4.7.1 Module: fussy.cronlock

Provide a cron lock for preventing cron-bombs and the like

exception `fussy.cronlock.`**`Busy`**
Raised if the lock is held by another cronlock

__weakref__
list of weak references to the object (if defined)

class fussy.cronlock.**Flock**(*filename*)
A context manager that just flocks a file

__weakref__
list of weak references to the object (if defined)

class fussy.cronlock.**Lock**(*lockfile*, *quiet_fail=False*)
A Context manager that provides cron-style locking

__init__(*lockfile*, *quiet_fail=False*)
Create a lock file

name – used to construct the lock-file name directory – directory in which to construct the lock-file

__weakref__
list of weak references to the object (if defined)

set_timeout(*duration*)
Set a signal to fire after duration and raise an error

exception fussy.cronlock.**Timeout**
Raised if the timeout signal triggers

__weakref__
list of weak references to the object (if defined)

fussy.cronlock.**with_lock**(*name*, *directory=None*, *timeout=None*)
Decorator that runs a function with Lock instance acquired

- name – basename of the file to create
- **directory – if specified, the directory in which to store files,** defaults to tempfile.gettempdir()
- **timeout – if specified, the number of seconds to allow before raising** a Timeout error

4.8 South Rollbacks

Note: fussy-southrollback should likely be considered a proof-of-concept or a bit of sample code. Your own migrations will likely require system specific modifications that would make using this script as-is an unreasonable choice.

If your firmware uses the South migration mechanism for Django, then you will likely need to be able to support running reverse migrations when the customer installs an older firmware. Because of how the migrations are stored, you will need to run the rollback migrations while the *old* firmware is installed.

You can support this by adding a call to fussy-southrollback to your `.pre-install` script. It will:

- scan the relative path (-p argument) for migrations
- do a textual compare on the sorted lists of migrations
- find the set of identical items
- **IFF the new migrations are *not* a superset of the current migrations**
  - will perform a django-admin.py migrate –noinput app migration
  - where migration is the last common migration among the pair of migrations

Note: You *must* not rename a migration if you are using this code.

4.8.1 Module: fussy.southrollback

Utility functions for common tasks

`fussy.southrollback.`**`find_reverse_upgrades`**(*final_target*, *relative_path*, *migration_pattern='????_*.py'*)

Find reverse upgrades that must be run before final_target is installed

final_target – final target of the installation relative_path – path from final target to Django/South migrations

returns ID/number of the migration to run (if any)

`fussy.southrollback.`**`get_options`**()

Creates the OptionParser used in `main()`

`fussy.southrollback.`**`main`**()

Finds last common migration and reverts current Django db to that

Must be run with DJANGO_SETTINGS_MODULE set in the environment

CHAPTER

FIVE

INDICES AND TABLES

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

PYTHON MODULE INDEX

f