cfengine concepts

Edition 2.2.1 for version 2.2.1

Mark Burgess
Faculty of Engineering, Oslo University College, Norway

This manual corresponds to CFENGINE Edition 2.2.1 for version 2.2.1 as last updated 3 March 2007.

1 Overview

In this manual the word "host" is used to refer to a single computer system – i.e. a single machine which has a name termed its "hostname".

1.1 What is cfengine and who can use it?

Cfengine is a tool for setting up and maintaining computer systems. It consists of several components:

cfagent An autonomous configuration agent (required).

cfservd A file server and remote activation service (optional).

cfexecd A scheduling and report service (recommended).

cfenvd An anomaly detection service (strongly recommended).

cfrun A way of activating cfagent remotely (use this as you need to).

cfshow A way of examining the contents of helper databases (helper).

cfenvgraph
Ancillary tool for cfenvd (helper).

cfkey Key generation tool (run once on every host).

The agent '`cfagent`' can be used without the other programs, but not all of the capabilities of cfengine will be available unless the components are installed and used appropriately.

Cfengine incorporates a declarative language—much higher level than Perl or shell: a single statement can result in many hundreds of operations being performed on multiple hosts. Cfengine is good at performing a lot of common system administration tasks, and allows you to build on its strengths with your own scripts. You can also use it as a netwide front-end for `cron`. Once you have set up cfengine, you'll be free to use your time doing other things instead of manual configuration.

The main purpose of cfengine is to allow you to create a single, central system configuration which will define how every host on your network should be configured, and to do so in an intuitive way. An interpreter runs on every host on your network and parses the master file (or file set). The configuration of each host is checked against this file; then, if you request it, any deviations from the defined configuration are fixed automatically. You do not have to mention every host specifically by name in order to configure them: instead you can refer to the properties which distinguish hosts from one another. Cfengine uses a flexible system of "classes" which helps you to single out a specific group of hosts with a single statement.

Cfengine grew out of the need to control the accumulation of complex shell scripts used in the automation of key system maintenance at University College in Oslo. There were a lot of scripts, written in shell and in Perl, performing system administration tasks such as file tidying, find-database updates, process checking and several other tasks. In a mixed environment, shell scripts work very poorly: shell commands have differing syntax across different operating systems, and the locations and names of key files differ. In fact, the non-uniformity of Unix was a major headache. Scripts were filled with tests to determine what kind of operating system they were being run on, to the point where they became so complicated and unreadable that no one was quite sure what they did anymore. Other scripts were placed only on the systems where they were relevant, out of sight and out of mind. It quickly became clear that our dream solution would be to replace this proliferation of scripts by a single file containing everything to be checked on every host on the network. By defining a new language, this file could hide all of the tests by using classes (a generalized 'switch/case' syntax) to label operations and improve the readability greatly. The gradual refinement of this idea resulted in the present day cfengine.

As an inexperienced cfengine user, you will probably find yourself trying to do things as you would have tried to do them in shell or Perl. This is probably not the right way to think when using cfengine. You will need to think in a more 'cfengine way'. When reading the manual, keep in mind that cfengine's way of working is to think about what the final result should be like, rather than on how to get there (with shell and Perl you specify what to do, rather than what you would like).

The remainder of this manual assumes that you know a little about BSD and UNIX System V systems and have every day experience in using either the C shell (csh) or the Bourne shell (sh), or their derivatives. If you are experienced in system administration, you might like to skip the earlier chapters and turn straight to the example in the section *Example configuration file* of the Reference manual. This is the probably quickest way to learn cfengine for the initiated. If you are not so familiar with system administration and would like a more gentle introduction, then we begin here...

1.2 Site configuration

To the system administrator of a small network, with just a few workstations or perhaps even a single mainframe system, it might seem superfluous to create a big fuss about the administration of the system. After all, it's easy to 'fix' things manually should any problems arise, making a link here, writing a script there and so on — and its probably not even worth writing down what you did because you know that it will always be easy to fix next time around too... But networks have a tendency to expand and—before you know it—you have five different types of operating system and each type of system has to be configured in a special way, you have to make patches to each system and you can't remember whether you fixed that host on the other side of the building... Also, you discover fairly quickly that what you thought of as BSD or System V is not as standard as you thought and that none of your simple scripts that worked on one system work on the others without a considerable amount of hacking and testing. You try writing a script to help you automate the task, but end up with an enormous number of '`if..then..else..`' tests which make it hard to see what is really going on.

To manage a network with many different flavours of operating system in a systematic way, what is needed is a more disciplined way of making changes which is robust against system re-installation. After all, it would be tragic to spend many hours setting up a system by hand only to lose everything in an unfortunate disk crash a week or even a year later when you have forgotten what you had to do. Upgrades of the operating system software might delete your carefully worked out configuration. What is needed is a separate record of all of the patches required on all of the systems on the network; a record which can be compared to the state of each host at any time and which a suitable engine can use to fix any deviations from that reference standard.

The idea behind cfengine is to focus upon a few key areas of basic system administration and provide a language in which the transparency of a configuration program is optimal. It eliminates the need for lots of tests by allowing you to organize your network according to “classes”. From a single configuration file (or set of files) you can specify how your network should be configured — and cfengine will then parse your file and carry out the instructions, warning or fixing errors as it goes.

1.3 Key Concepts

Some of the important issues in system administration which cfengine can help with.

1.3.1 Configuration files and registries

One of the endearing characteristics of BSD and System V systems is that they are configured through human-readable text files. To add a new user to the system you edit ‘`/etc/passwd`’, to add a new disk you must edit ‘`/etc/fstab`’, etc. Many applications are also configured with the help of text files. When installing a new system for the first time, or when changing updating the setup of an old system, you are faced with having to edit lots of files. In some cases you will have to add precisely the same line to the same file on every system in your network as a change is made, so it is handy to have a way of automating this procedure so that you don’t have to load every file into an editor by hand and make the changes yourself. This is one of the tasks which cfagent will automate for you.

On Windows systems, configuration data are stored in a system registry. With the right tools, the Windows system registry can also be edited by cfengine, but this requires more care.

1.3.2 Network interface

Each host which you connect to an Ethernet-based network running TCP/IP protocols must have a so-called ‘net interface’. This network interface must be configured before it will work. Normally, one does this with the help of the `ifconfig` command. This can also be checked and configured automatically by cfagent.

Network configuration involves telling the interface hardware what the internet (IP) address of your system is, so that it knows which incoming ‘packets’ of data to pay attention to. It involves telling the interface how to interpret the addresses it receives by setting the ‘netmask’ for your network (see below). Finally, you must tell it which dummy address is to be used for messages which are broadcast to all hosts on your network simultaneously (see the Reference Manual).

Cfagent’s features are mainly meant for hosts which use static IP addresses; if you are using DHCP clients, then you will not need the net configuration features.

1.3.3 Network File System (NFS) or file distribution?

Probably the first thing you are interested in doing with a network (after you’ve had your fill of the World-Wide Web) is to make your files available to some or all hosts on the network, no matter where in your corporate empire (or university dungeon) you might be sitting. In other words, if you have a disk which is physically connected to host A, you would like to make the contents of that disk available to hosts B, C, D ..., etc. NFS (the Network FileSystem) makes this possible.

The process works by 'filesystems'. A filesystem is one partition of a disk drive – or one unit of disk space which can be accessed by a single 'logical device' '/dev/something'.

To make a filesystem available to other hosts you have to do three things.

- On the host the disk is physically connected to you must *export* the filesystem by adding something to the file '/etc/exports'. This tells NFS who is allowed to access the disk and who isn't.
- On the host which is to access the filesystem you must create a mount point. This is a name in the directory tree at which you want to add the files to your local filesystem.
- On the host which is to access the files you must mount the filesystem onto the mount point. The mount operation is the jargon for telling the system to access the device on which the data are stored. Mounting is analogous to opening a file: files are opened, filesystems are mounted.

Only after all three of these have been done will a filesystem become available across the network. Cfagent will help you with the last two in a very transparent way. You could also use the text-editing facility in cfagent to edit the 'exports' file, but there are other ways to update the 'exports' file using NIS and *netgroups*, which we shall not go into here. If you are in doubt, look up the manual page on exports(5).

Some sites prefer to minimize the use of NFS filesystems to avoid one machine being dependent on another. They prefer to make a local copy of the files on a remote machine instead. Traditionally, programs like `rdist` have been used for this purpose. You may also use cfagent to copy files in this way, See Section 6.2.1 [Emulating rdist], page 63.

1.3.4 Name servers (DNS)

There are two ways to specify addresses on the internet (called IP addresses). One is to use the text address like 'ftp.uu.net' and the other is to use the numerical form '192.48.96.9'. Alas, there is no direct one-to-one correspondence between the numerical addresses and the textual ones, thus a service (called DNS) is required to map one to the other.

The service is performed by one or more special hosts on the network called *name servers*. Each host must know how to contact a name server or it will probably hang the first time you give it an IP address. You tell it how to contact a name server by editing the text-file '/etc/resolv.conf'. This file must contain the domain name for your domain and a list of possible name servers which can be contacted, in order of priority. Since this is a special file which every host must have, you don't have to use the general text file editing facilities in cfagent. You can just define the name servers for each host in the cfagent file and cfagent will do the editing to '/etc/resolv.conf' automatically. If you want to change the priority of name servers later, or even change the list then a simple change of one or two lines in the configuration file will enable you to reconfigure every host on your network automatically without having to do any editing yourself!

1.3.5 Monitoring important files

Security is an important issue on any system. In the busy life of a system administrator, it is not always easy to remember to set the correct access rights on every file; this can result in either a security breach or problems in accessing files.

A common scenario is that you, as administrator, fetch a new package using FTP, compile it and install it without thinking too carefully. Since the owner and permissions of the files in an FTP archive remains those of the program author, it often happens that the software is left lying around with the owner and permissions as set by the author of the program rather than any user name on *your* system. The userid of the author might be anybody on your system — or perhaps nobody at all! The files should clearly be owned by root and made readable but unwritable by normal users.

Simple accidents and careless actions under stress could result in, for example, the password file being writable to ordinary users. If this were the case, the security of the entire system would be compromised. Cfagent therefore allows you to monitor the permissions, ownership, and general existence of files and directories and, if you wish, to either correct them or warn about them automatically.

1.3.6 Making links

One of the difficulties with having so many different variations on the theme of BSD and System V based operating systems is that similar files are not always where you expect to find them. They have different names or lie in different directories. The usual solution to the problem is to make an alias for these files, or a pointer from one filename to another. The name for such an alias is a *symbolic link*.

It is often very convenient to make symbolic links. For example, you might want the sendmail configuration file '`/etc/sendmail.cf`' to be a link to a global configuration file, say,

    '/usr/local/mail/etc/sendmail.cf'

on every single host on your network so that there is only one file to edit. If you had to make all of these links yourself, it would take a lifetime. Cfagent will make such a link automatically and check it each time it is run. You can also ask it to tidy up old links which have been left around and no longer point to existing files. If you reinstall your operating system later, it doesn't matter because all your links are defined in your cfagent configuration file, recorded for all time. Cfengine won't forget it, and you won't forget it because the setup is defined in one central place.

Cfagent will also allow you to make hard links to regular files, but not to other kinds of files. A hard link that points to a symbolic link is the same as a hard link to the file the symbolic link points to.

1.4 Functionality

The notes above give you a rough idea of what cfengine can be used for. Here is a quick summary of cfagent's capabilities.

- Check and configure the network interface.
- Edit textfiles for the system and for all users.
- Make and maintain symbolic links, including multiple links, using a single command.
- Check and set the permissions and ownership of files.
- Tidy (i.e., delete) junk files which clutter the system.
- Systematic and automated mounting of NFS filesystems.
- Checking for the presence of important files and filesystems.
- Controlled execution of user scripts and shell commands.

- A class-based decision structure.
- Process management.

How do you run cfagent? You can run it as a cron job, or you can run it manually. You may run cfagent scripts/programs as often as you like. Each time you run a script, cfengine determines whether anything needs to be done — if nothing needs to be done, nothing is done! If you use it to monitor and configure your entire network from a central file-base, then the natural thing is to run cfengine repeatedly with the help of `cron` and/or `cfexecd`.

2 Getting started

2.1 What you must have in a cfagent program

A cfagent configuration file for a large network can become long and complex so, before we get down to details, let's try to strip away the complexity and look only to the essentials.

Each cfagent program or configuration file is a list of declarations of items to be checked and perhaps fixed. You begin by creating a file called '`cfagent.conf`'. The simplest meaningful file you can create is something like this:

```
# Comment...

control:

  actionsequence = ( links )

links:

  /bin -> /usr/bin
```

The example above checks and makes (if necessary) a link from '`/bin`' to '`/usr/bin`'. Let's examine this example more closely. In a cfengine program:

- Use of space is unrestricted. You can start new lines wherever you like. You should generally have a space before and after parentheses to avoid confusing the parser.
- A comment is some text which is ignored by cfengine. The '`#`' symbol designates a comment and means: ignore the remaining text on this line. A comment symbol must have a space in front of it, or start a new line so that cfengine knows you don't mean the symbol as part of another word.
- Words which end in a single colon define *sections* in a program. Under a given section you group together all declarations of a given type. Section names must all be taken from a list defined by the language. You cannot define your own sections.
- Words which end in two colons are so-called *class* names. They are used for making decisions in cfengine.
- Statements which are of the form `name=( list )` are used to assign the value on the right hand side to the name on the left hand side of the equals sign.

In simple example above has three of the four types of object described above. The `control:` section of any program tells cfengine how to behave. In this example it adds the action *links* to the action sequence. For *links* you could replace some other action. The essential point is that, if you don't have an action sequence, your cfengine program will do absolutely nothing! The action sequence is a list which tells cfagent what do to and in which order.

The `links:` section of the file tells cfagent that what follows is a number of links to be made. If you write this part of the file, but forget to add links to the action sequence, then nothing will be done! You can add any number of links in this part of the file and they will all be dealt with in order when—and only when—you write *links* in the action sequence.

To summarize, you *must* have:

- Some declarations which specify things to be done.

- An action sequence which tells cfagent which sections to process, how many times and in which order they should be processed.

Now let's think a bit about how useful this short example program is. On a SunOS (Solaris) system, where the directory '`/bin`' is in fact supposed to be a link, such a check could be useful, but on some other system where '`/bin`' is a not a link but a separate directory, this would result in an error message from cfagent, telling you that '`/bin`' exists and is not a link. The lesson is that, if we want to use cfagent to make *one single* program which can be run on any host of any type, then we need some way of restricting the above link so that it only gets checked on SunOS systems. We can write the following:

```
# Comment...

control:

  actionsequence = ( links  )

links:

  sun4::

       /bin -> /usr/bin
       # other links

   osf::

       # other links
```

The names which have double colons after them are called *classes* and they are used to restrict a particular action so that it only gets performed if the host running the program is a member of that class. If you are familiar with C++, this syntax should make you think of classes definitions in C++. Classes works like this: the names above `sun4`, `sun3`, `osf` etc. are all internally defined by cfagent. If a host running, say, the OSF operating system executes the file it automatically becomes a member of the class `osf`. Since it cannot be a member more than one of the above, this distinguishes between different types of operating system and creates a hidden `if`..`then`...`else` test.

This is the way in which cfagent makes decisions. The key idea is that actions are only carried out if they are in the same class as the host running the program. Classes are dealt with in detail in the next chapter.

Now let's see how to add another kind of action to the action sequence.

```
# Comment...

control:

  actionsequence = ( tidy links )

links:

  /bin -> /usr/bin

tidy:
```

```
/tmp  pattern=* age=7 recurse=inf
```

We have now added a new kind of declaration called `tidy:` which deletes files. In the example above, we are looking at files in the directory '`/tmp`' which match the pattern '`*`' and have not been accessed for more than seven days. The search for these files descends recursively down any number of subdirectories.

To make any of this happen we must add the word *tidy* to the action sequence. If we don't, the declaration will be ignored. Notice also that, regardless of the fact that `links:` comes before `tidy:`, the order in the action sequence tells us that all `tidy` actions will be performed before `links:`.

The above structure can be repeated to build up a configuration file or script.

2.2 Program structure

To summarize the previous section, here is a sketch of a typical cfagent configuration program showing a sensible structure. The various sections are listed in a sensible order which you would probably use in the action sequence.

An individual section-declaration in the program looks something like this:

```
action-type:

   class1::

       list of things to do...

   class2::

       list of things to do...
```

`action-type` is one of the following reserved words:

```
groups, control, copy, homeservers, binservers, mailserver, mountables,
import, broadcast, resolve, defaultroute, directories, miscmounts,
files, ignore, tidy, required, links, disable, shellcommands, strategies
editfiles, processes
```

The order in which declarations occur is not important to cfengine from a syntactical point of view, but some of the above actions define information which you will want to refer to later. All variables, classes, groups etc. must be defined before they are used. That means that it is smart to follow the order above for the sections in the first line of the above list.

The order in which items are declared is not to be confused with the order in which they are executed. This is determined by the `actionsequence`, (see the reference manual). Probably you will want to coordinate the two so that they match as far as possible.

For completeness, here is a complete summary of the structure of a very general cfagent configuration program. The format is free and use of space is unrestricted, though it is always a good idea to put a space in front before and after parentheses when defining variables.

```
######################################################################
#
```

```
# Example of structure
#
######################################################################

groups:

   group1 = ( host host ...  )
   group2 = ( host host ...  )
   ...

######################################################################

control:

   class::

   site      =  ( mysite )
   domain    =  ( mydomain )
   ...

    actionsequence =
      (
      action name
      ....
      )

   mountpattern = ( mountpoint )
   homepattern = ( wildcards matching home directories )

   addinstallable = ( foo bar )
   addclasses     = ( foo bar )

######################################################################

homeservers:

   class::
           home servers

binservers:

   class::
           binary servers

mailserver:

   class::
           mail server

mountables:

   class::

           list of resources


######################################################################
```

```
import:

   class::    include file

   class::    include file

######################################################################

broadcast:

  class::  ones   # or zeros / zeroes

defaultroute:

   class::  my-gw

######################################################################

resolve:

   any::

       list of nameservers


   ...
```

2.3 Building a distributed configuration

If a configuration is to be specified at one central location, how does it get distributed to many hosts? The simple answer is to get cfengine to distribute the configuration to the hosts. To do that, a separate configuration file is used. Why?

Imagine what would happen if you made a mistake in the configuration, i.e. a syntax error which got distributed to every host. Now all the hosts would be unable to run cfengine, and thereafter unable to download a corrected configuration file. The whole setup would be broken. To prevent this kind of accident, a separate configuration file is used to copy the files and binaries to each host. This configuration should be simple, and should almost never be edited: they key word here is *reliability*.

2.3.1 Startup update.conf

The file '`update.conf`' can have more or less the same form for all sites, looking something like this.

```
#######
#
# BEGIN update.conf
#
# This script distributes the configuration, a simple file so that,
# if there are syntax errors in the main config, we can still
# distribute a correct configuration to the machines afterwards, even
# though the main config won't parse. It is read and run just before the
# main configuration is parsed.
```

```
#
#######

control:

 actionsequence  = ( copy tidy )  # Keep this simple and constant

 domain          = ( iu.hio.no )  # Needed for remote copy

 #
 # Which host/dir is the master for configuration roll-outs?
 #

 policyhost      = ( nexus.iu.hio.no )
 master_cfinput  = ( /masterfiles/inputs )

 #
 # Some convenient variables
 #

 workdir         = ( /var/cfengine )
 cf_install_dir  = ( /usr/local/sbin )

 # Avoid server contention

 SplayTime = ( 5 )

#############################################################################

 #
 # Make sure there is a local copy of the configuration and
 # the most important binaries in case we have no connectivity
 # e.g. for mobile stations or during DOS attacks
 #

copy:

     $(master_cfinput)            dest=$(workdir)/inputs
                                  r=inf
                                  mode=700
                                  type=binary
                                  exclude=*.lst
                                  exclude=*~
                                  exclude=#*
                                  server=$(policyhost)

     $(cf_install_dir)/cfagent    dest=$(workdir)/bin/cfagent
                                  mode=755
                                  backup=false
                                  type=checksum

     $(cf_install_dir)/cfservd    dest=$(workdir)/bin/cfservd
                                  mode=755
                                  backup=false
                                  type=checksum

     $(cf_install_dir)/cfexecd    dest=$(workdir)/bin/cfexecd
                                  mode=755
```

```
                                  backup=false
                                  type=checksum

######################################################################

tidy:

     #
     # Cfexecd stores output in this directory.
     # Make sure we don't build up files and choke on our own words!
     #

     $(workdir)/outputs pattern=* age=7

#######
#
# END cf.update
#
#######
```

2.3.2 Startup cfservd.conf

In order to set up remote distribution from a central server, you will need to start the cfservd service on the host from which the configuration is to be copied, and grant access to the hosts which need to download it. Here is a simple get-started file which does this:

```
#########################################################
#
# This is a cfservd config file - it is used for the server
# part of cfengine, for remote file transfers and control
# over cfengine using the cfrun program.
#
#########################################################

control:

  domain = ( iu.hio.no )

     cfrunCommand = ( "/var/cfengine/bin/cfagent" )

 any::

  IfElapsed = ( 1 )
  ExpireAfter = ( 15 )
  MaxConnections = ( 50 )
  MultipleConnections = ( true )

#########################################################

grant:

   # Grant access to all hosts at example.org.
   # Files should be world readable

   /masterfiles/inputs        *.example.org

   # Make sure there is permission to execute by cfrun
```

```
   /var/cfengine/bin/cfagent  *.example.org

########
#
# END cfservd.conf
#
########
```

2.3.3 Where should I put the files?

Where should the files be located? To organize your files, you should think of three potential locations, for different purposes:

- A version controlled repository for authoring, testing and applying changes to the files. Once a version is approved for release, these master files should be moved on to a publishing area. e.g. '`/usr/local/masterfiles/cfengine/inputs`' on 'masterhost'.
- A centralized location, into which you publish your altered and tested configurations. This is the 'buffer' location from which each client will download new and tested versions of the master configuration. e.g. '`/usr/local/masterfiles/cfengine/inputs`' on 'masterhost'. It acts as a buffer between the testing ground and the production location.
- The Work Directory. This is the production location. It is a location that is normally chosen to be the private directory where cfagent expects to find its configuration files, '`/var/cfengine/inputs`'. Your '`update.conf`' file has the job of copying from the master location (second bullet) above to this location.

Modules and methods are normally kept in a separate directory than inputs files are kept in, because they require a directory with special authorizations whe executing. This is good practice As long as the '`update.conf`' places the master versions in the correct location (usually '`/var/cfengine/modules`') on the local host, all will be okay.

You should not try to copy files directly from a version controlled repository, as you might end up sending out an incomplete or partially tested version of the files to all your hosts.

```
# Example update.conf

control:

   master_cfinput  = ( /usr/local/masterfiles/cfengine/inputs )
   workdir         = ( /var/cfengine )

copy:

   # Copy from bullet 2 to bullet 3

     $(master_cfinput)            dest=$(workdir)/inputs
                                  r=inf
                                  mode=700
                                  type=binary
                                  exclude=*.lst
                                  exclude=*~
                                  exclude=#*
                                  server=$(policyhost)
                                  trustkey=true
```

```
     $(master_modules)            dest=$(workdir)/modules
                                  r=inf
                                  mode=700
                                  type=binary
                                  exclude=*.lst
                                  exclude=*~
                                  exclude=#*
                                  server=$(policyhost)
                                  trustkey=true
```

2.4 Optional features in cfagent

Cfagent doesn't do anything unless you ask it to. When you run a cfagent program it generates no output unless it finds something it believes to be wrong. It does not carry out any actions unless they are declared in the action sequence.

If you like, though, you can make cfagent positively chatty. Cfagent can be run with a number of command line options (see the reference manual). If you run the program with the '-v' or '--verbose' options, it will supply you cheerily with a resume of what it is doing. Certain warning messages also get printed in verbose mode, so it is a useful debugging tool.

You can ask cfagent to check lots of things – the timezone for instance, or the domain name. In order for it to check these things, it needs some information from you. All of the switches and options which change the way in which cfagent behaves get specified either on the command line or in the `control:` section of the control file. Some special control variables are used for this purpose. Here is a short example:

```
control:

  domain   = ( example.org )
  netmask  = ( 255.255.255.0 )
  timezone = ( MET CET )

  mountpattern = ( /mydomain/mountpoint )

  actionsequence =
     (
     checktimezone     # check time zone
     netconfig         # includes check netmask
     resolve           # includes domain
     mountinfo         # look for mounted disks under mountpattern
     )
```

To get verbose output you must run cfagent with the appropriate command line option '--verbose' or '-v'.

Notice that setting values has a special kind of syntax: a variable name, an equals sign and a value in parentheses. This tells you that the quantity of the left hand side assumes the value on the right hand side. There are lots of questions you might ask at this point. The answers to these will be covered as we go along and in the next chapter.

Before leaving this brief advertisement for control parameters, it is worth noting the definition of `mountpattern` above. This declares a directory in which cfagent expects to find mounted disks. It will be explained in detail later, for now notice that this definition looks rather stupid and inflexible. It would be much better if we could use some kind of variables to define where to look for mounted filesystems. And of course you can...

Having briefly scraped the surface of what cfagent can do, turn to the example and take a look at what a complete program can look like, (see the reference manual). If you understand it, you might like to skip through the rest of the manual until you find what you are looking for. If it looks mysterious, then the next chapter should answer some questions in more depth.

2.5 Invoking cfagent

Cfagent may be invoked in a number of ways. Here are some examples:

```
host% cfagent

host% cfagent --file myfile

host% cfagent -f myfile -v -n

host% cfagent --help
```

The first of these (the default command, with no arguments) causes cfagent to look for a file called '`cfagent.conf`' in the directory pointed to by the environment variables `CFINPUTS` or '`/var/cfengine/inputs`' by default, and execute it silently. The second command reads the file '`myfile`' and works silently. The third works in verbose mode and the `-n` option means that no actions should actually be carried out, only warnings should be printed. The final example causes cfagent to print out a list of its command line options.

The complete list of options is listed in the summary at the beginning of this manual, or you can see it by giving the `-h` option, (see the reference manual).

In addition to running cfagent with a filename, you can also treat cfagent files as scripts by starting your cfagent program with the standard shell line:

```
#!/usr/local/sbin/cfagent -f
#
# My config script
#
```

Here we assume that you have installed cfengine under the directory '`/usr/local/sbin`'. By adding a header like this to the first line of your program and making the file executable with the `chmod` shell command, you can execute the program just by typing its name—i.e. without mentioning cfengine explicitly at all.

As a novice to cfengine, it is advisable to check all programs with the `-n` option before trusting them to your system, at least until you are familiar with the behaviour of cfengine. This 'safe' option allows you to see what cfengine wants to do, without actually committing yourself to doing it.

2.6 Running cfengine permanently, monitoring and restarting cfexecd

Once you are happy using cfengine, you will want it to run least once per hour on your systems. This is easily achieved by adding the following line to the root crontab file of each system:

```
0,30 * * * * /usr/local/sbin/cfexecd -F
```

This is enough to ensure that cfengine will get run. Any output generated by this job, will be stored in '/var/cfengine/outputs'. In addition, if you add the following to the file 'cfagent.conf', the system administrator will be emailed a summary of any output:

```
control:

smtpserver = ( mailhub.example.org ) # site MTA which can talk smtp
sysadm     = ( mark@example.org )   # mail address of sysadm
```

Fill in suitable values for these variables. An alternative, or additional way to run cfengine, is to run the 'cfexecd' program is daemon mode (without the '-F') option. In this mode, the daemon lives in the background and sleeps, activating only in accordance with a scheduling policy. The default policy is to run once every hour (equivalent to Min00_05). Here is how you would modify 'cfagent.conf' in order to make the daemon execute cfagent every half-hour:

```
 control:

   # When should cfexecd in daemon mode wake up the agent?

   schedule   = ( Min00_05 Min30_35 )
```

Note that the time specifications are the basic cfengine *time classes*, See Section 5.3 [Building flexible time classes], page 60. Although one of these methods should suffice, no harm will arise from running both cron and the cfexecd side-by-side. Cfagents locking mechanisms ensure that no contention will occur.

The other components of cfengine can be started by cfagent itself:

```
processes:

 "cfenvd"  restart "/usr/local/sbin/cfenvd"
 "cfservd" restart "/usr/local/sbin/cfservd"
```

Note that, to start cfexecd by cfengine, one must do this

```
processes:

 "bin/cfexecd$"  restart "/usr/local/sbin/cfexecd"
```

It's important to use as specific a regular expression as possible in match statements (the path to the program and the regular expression metacharacter `$` meaning "end of line", in this example) because bare strings can often match unexpected processes. For instance, using `cfexecd` by itself will also match a process spawned by `cfagent -F`, which shows up as `/var/cfagent/bin/cfagent -Dfrom_cfexecd` in the process table!

2.7 CFINPUTS environment variable

Whenever cfengine looks for a file it asks a question: is the filename an absolute name (that is a name which begins from '`/`' like `/usr/file`), is it a file in the directory in which you invoke cfengine or is it a file which should be searched for in a special place?

If you use an absolute filename either on the command line using `-f` or in the `import` section of your program (a name which begins with a slash '/'), then cfengine trusts the name of the file you have given and treats it literally. If you specify the name of the file as simple '`.`' or '`-`' then cfengine reads its input from the standard input.

If you run cfengine without arguments (so that the default filename is '`cfagent.conf`') or you specify a file without a leading slash in the `import` section, then the value of the environment variable `CFINPUTS` is prepended to the start of the file name. This allows you to keep your configuration in a standard place, pointed to by `CFINPUTS`. For example:

```
host# setenv CFINPUTS /usr/local/masterfiles/cfengine/inputs

host# cfagent -f myfile
```

In this example, cfengine tries to open '`myfile`'. in the directory '`/usr/local/masterfiles/cfengine/inputs`'. If no value is set for `CFINPUTS`, then the default location is the trusted cfengine directory '`/var/cfengine/inputs`'.

2.8 What to aim for

If you are a beginner to cfengine, you might not be certain exactly how you want to use it. Here are some hints from Dr. Daystrom about how to get things working quickly.

- Run cfengine from cron every hour on all your systems. Be sure to label long tasks, or tasks which do not need to be performed often by a *time class* which prevents it from being executed all the time, See Chapter 5 [Using cfengine as a front-end for cron], page 59.

  Running cfengine from cron means that it will be run in parallel on your systems. Cfengine on one host does not have to wait for cfengine on another host to complete.

- Set up `cfservd` on all your systems so that cfengine can be executed remotely, so that you can immediately "push" changes to all your hosts with `cfrun`. Think carefully about whom you wish to give permission to run cfengine from the net, See Section 6.3 [Configuring cfservd], page 68. Set up you '`cfservd.conf`' file accordingly. You can also use this daemon to grant access rights for remote file copying.

  Cfrun polls all your hosts serially and gives you a concatenated indexed list of problems on all hosts. The disadvantage with cfrun is that each host has to wait its turn.

- Don't forget to add `cfservd` to the system startup scripts, or to '`inittab`' so that it starts when you boot your system.

- Add *all* your hosts to the '`cfrun.hosts`' file. It does not matter that some may be master servers and others clients. The locking mechanisms will protect you from silliness, See Section 6.2.5 [Deadlocks and runaway loops], page 67. Cfengine will work it out. Cfrun allows you to remotely execute cfengine on groups of hosts which satisfy a list of cfengine classes.

When you have set up these components, you can sit back and edit the configuration files and watch things being done.

3 More advanced concepts

3.1 Classes

The idea of classes is central to the operation of cfengine. Saying that cfengine is 'class orientated' means that it doesn't make decisions using `if`...`then`...`else` constructions the way other languages do, but only carries out an action if the host running the program is in the same class as the action itself. To understand what this means, imagine sorting through a list of all the hosts at your site. Imagine also that you are looking for the *class* of hosts which belong to the computing department, which run GNU/Linux operating system and which have yellow spots! To figure out whether a particular host satisfies all of these criteria you first delete all of the hosts which are not GNU/Linux, then you delete all of the remaining ones which don't belong to the computing department, then you delete all the remaining ones which don't have yellow spots. If you are on the remaining list, then you are in the class of all computer-science-Linux-yellow-spotted hosts and you can carry out the action.

Cfengine works in this way, narrowing things down by asking if a host is in several classes at the same time. Although some information (like the kind of operating system you are running) can be obtained directly, clearly, to make this work we need to have lists of which hosts belong to the computer department and which ones have yellow spots.

So how does this work in a cfengine program? A program or configuration script consists of a set of declarations for what we refer to as *actions* which are to be carried out only for certain classes of host. Any host can execute a particular program, but only certain action are extracted — namely those which refer to that particular host. This happens automatically because cfengine builds up a list of the classes to which it belongs as it goes along, so it avoids having to make many decisions over and over again.

By defining classes which classify the hosts on your network in some easy to understand way, you can make a single action apply to many hosts in one go – i.e. just the hosts you need. You can make generic rules for specific type of operating system, you can group together clusters of workstations according to who will be using them and you can paint yellow spots on them – what ever works for you.

A *cfengine action* looks like this:

```
action-type:

   compound-class::

       declaration
```

A single class can be one of several things:

- The name of an operating system architecture e.g. `ultrix`, `sun4` etc. This is referred to henceforth as a *hard class*.
- The (unqualified) name of a particular host. If your system returns a fully qualified domain name for your host, cfagent truncates it so as to unqualify the name.
- The name of a user-defined group of hosts.
- A day of the week (in the form `Monday`, `Tuesday`, `Wednesday`...).

- An hour of the day (in the form `Hr00`, `Hr01`, ... `Hr23`).
- Minutes in the hour (in the form `Min00`, `Min17`, ... `Min45`).
- A five minute interval in the hour (in the form `Min00_05`, `Min05_10`, ... `Min55_00`)
- A quart hour (in the form `Q1, Q2, Q3, Q4`)
- An abbreviated time with quarter hour specified (in the form `Hr00_Q1`, `Hr23_Q4`, etc.)
- A day of the month (in the form `Day1 ... Day31`).
- A month (in the form `January`, `February`, ... `December`).
- A year (in the form `Yr1997`, `Yr2001`).
- An arbitrary user-defined string. (see the reference manual).
- The ip-address octets of any active interface (in the form `ipv4_192_0_0_1`, `ipv4_192_0_0`, `ipv4_192_0,ipv4_192`).

A compound class is a sequence of simple classes connected by dots or 'pipe' symbols (vertical bars). For example:

```
myclass.sun4.Monday::

sun4|ultrix|osf::
```

A compound class evaluates to 'true' if all of the individual classes are separately true, thus in the above example the actions which follow `compound_class::` are only carried out if the host concerned is in `myclass`, is of type `sun4` and the day is Monday! In the second example, the host parsing the file must be either of type `sun4` *or* `ultrix` *or* `osf`. In other words, compound classes support two operators: AND and OR, written '`.`' and '`|`' respectively. From cfengine version 2.1.1, I bit the bullet and added '`&`' as a synonym for the AND operator. Cfengine doesn't care how many of these operators you use (since it skips over blank class names), so you could write either

```
solaris|irix::
```

or

```
solaris||irix::
```

depending on your taste. On the other hand, the order in which cfengine evaluates AND and OR operations *does* matter, and the rule is that AND takes priority over OR, so that '`.`' binds classes together tightly and all AND operations are evaluated before ORing the final results together. This is the usual behaviour in programming languages. You can use round parentheses in cfengine classes to override these preferences.

Cfengine allows you to define switch on and off dummy classes so that you can use them to select certain subsets of action. In particular, note that by defining your own classes, using them to make compound rules of this type, and then switching them on and off, you can also switch on and off the corresponding actions in a controlled way. The command line options `-D` and `-N` can be used for this purpose. See also `addclasses` in the Reference manual.

A logical NOT operator has been added to allow you to exclude certain specific hosts in a more flexible way. The logical NOT operator is (as in C and C++) '`!`'. For instance, the following example would allow all hosts except for `myhost`:

```
action:

 !myhost::

     command
```

and similarly, so allow all hosts in a user-defined group `mygroup`, *except* for `myhost`, you would write

```
action:

 mygroup.!myhost::

     command
```

which reads 'mygroup AND NOT myhost'. The NOT operator can also be combined with OR. For instance

```
class1|!class2
```

would select hosts which were either in class 1, or those which were not in class 2.

Finally, there is a number of reserved classes. The following are hard classes for various operating system architectures. They do not need to be defined because each host knows what operating system it is running. Thus the appropriate one of these will always be defined on each host. Similarly the day of the week is clearly not open to definition, unless you are running cfengine from outer space. The reserved classes are:

```
ultrix, sun4, sun3, hpux, hpux10, aix, solaris, osf, irix4, irix, irix64
   sco, freebsd, netbsd, openbsd, bsd4_3, newsos, solarisx86, aos,
          nextstep, bsdos, linux, debian, cray, unix_sv, GnU, NT
```

If these classes are not sufficient to distinguish the hosts on your network, cfengine provides more specific classes which contain the name and release of the operating system. To find out what these look like for your systems you can run cfengine in 'parse-only-verbose' mode:

```
cfagent -p -v
```

and these will be displayed. For example, Solaris 2.4 systems generate the additional classes `sunos_5_4` and `sunos_sun4m`, `sunos_sun4m_5_4`.

Cfengine uses both the unqualified and fully host names as classes. Some sites and operating systems use fully qualified names for their hosts. i.e. `uname -n` returns to full domain qualified hostname. This spoils the class matching algorithms for cfengine, so cfengine automatically truncates names which contain a dot '.' at the first '.' it encounters. If your hostnames contain dots (which do not refer to a domain name, then cfengine will be confused. The moral is: don't have dots in your host names! *NOTE: in order to ensure that the fully qualified name of the host becomes a class you must define the domain variable.* The dots in this string will be replaced by underscores.

In summary, the operator ordering in cfengine classes is as follows:

'`()`' Parentheses override everything.

'`!`' The NOT operator binds tightest.

‘. &’ The AND operator binds more tightly than OR.

‘|’ OR is the weakest operator.

3.2 Variable substitution

When you are building up a configuration file it is very useful to be able to use variables. If you can define your configuration in terms of some key variables, it can be changed more easily later, it is more transparent to the reader of the program and you can also choose to define the variables differently on different types of system. Another way of saying this is that cfengine variables also belong to classes. Cfengine makes use of variables in three ways.

- Environment variables from the shell
- Special variables used in cfengine features
- General macro-string substitution.

Environment variables are fetched directly from the shell on whatever system is running the program. An example of a special variable is the `domain` variable from the previous section. Straightforward macro substitution allows you to define a symbol name to be replaced by an arbitrary text string. All these definitions (apart from shell environment variables, of course) are made in the control part of the cfengine program:

```
control:

  myvar = ( /usr/local/mydir/lib/very/long/path )   # define macro

...

links:

  $(myvar) -> /another/directory
```

Here we define a macro called `myvar`, which is later used to define the creation of a link. As promised we can also define class-dependent variables:

```
control:

  sun4:: myvar = ( sun )
  hpux:: myvar = ( HP )
```

Cfagent gives you access to the shell environment variables and allows you to define variables of your own. It also keeps a few special variables which affect the way in which cfengine works. When cfengine expands a variable it looks first at the name in its list of special variables, then in the list of user-defined macros and finally in the shell environment for a match. If none of these are found it expands to the empty string. If you nest macros,

```
control:

  myvar = ( "$(othervar)" )
```

then you must quote the right hand side and ensure that the value is already defined.

You can also import values from the execution of a shell command by prefixing a command with the word `exec`. This method is deprecated in version 2 and replaced by a function.

```
control:

 # old method

 listing = ( "exec /bin/ls -l" )

 # new method

 listing = ( ExecResult(/bin/ls -l) )
```

This sets the variable 'listing' to the output of the command in the quotes.

Some other internal functions are

`RandomInt(a,b)`
: Generate a random integer between a and b.

`ExecResult(`*command*`)`
: Executes the named shell command and inserts the output into the variable.

For example:

```
control:

 variable2 = ( RandomInt(0,23) )

 variable3 = ( ExecResult(/bin/ls -a /opt) )
```

Variables are referred to in either of two different ways, depending on your taste. You can use the forms `$(variable)` or `${variable}`. The variable in braces or parentheses can be the name of any user defined macro, environment variable or one of the following special internal variables.

`AllClasses`
: A long string in the form '`CFALLCLASSES=class1:class2...`'. This variable is a summary of all the defined classes at any given time. It is always kept up to date so that scripts can make use of cfengine's class data.

`arch`
: The current detailed architecture string—an amalgamation of the information from *uname*. Non-definable.

`binserver`
: The default server for binary data. See Section 4.6 [NFS resources], page 43. Non definable.

`class`
: The currently defined system hard-class (e.g. `sun4`, `hpux`). Non-definable.

`date`
: The current date string. Note that if you use this in a shell command it might be interpreted as a list variable, since it contains the default separator '`:`'.

`domain`
: The currently defined domain.

`faculty` The faculty or site as defined in control (see site).

`fqhost` The fully qualified (DNS/BIND) hostname of the system, which includes the domain name as well.

`host` The hostname of the machine running the program.

`ipaddress`
: The numerical form of the internet address of the host currently running cfengine.

`MaxCfengines`
: The maximum number of cfengines which should be allowed to co-exist concurrently on the system. This can prevent excessive load due to unintentional spamming in situations where several cfagents are started independently. The default value is unlimited.

`ostype` A short for of `$(arch)`.

`OutputPrefix`
: This quoted string can be used to change the default 'cfengine:' prefix on output lines to something else. You might wish to shorten the string, or have a different prefix for different hosts. The value in this variable is appended with the name of the host. The default is equivalent to,

```
OutputPrefix = ( "cfengine:$(host):")
```

`RepChar` The character value of the string used by the file repository in constructing unique filenames from path names. This is the character which replaces '`/`' (see the reference manual).

`site` This variable is identical to `$(faculty)` and may be used interchangeably.

`split` The character on which list variables are split (see the reference manual).

`sysadm` The name or mail address of the system administrator.

`timezone` The current timezone as defined in `control`.

`UnderscoreClasses`
: If this is set to 'on' cfengine uses hard-classes which begin with an underscore, so as to avoid name collisions. See also Runtime Options in the Reference manual.

`year` The current year.

These variables are kept special because they play a special role in setting up a system configuration. See Chapter 4 [Global configurations], page 39. You are encouraged to use them to define fully generalized rules in your programs. Variables can be used to advantage in defining filenames, directory names and in passing arguments to shell commands. The judicious use of variables can reduce many definitions to a single one if you plan carefully.

> *NOTE: the above control variables are not case sensitive, unlike user macros, so you should not define your own macros with these names.*

The following variables are also reserved and may be used to produce troublesome special characters in strings.

`cr` Expands to the carriage-return character.

`dblquote` Expands to a double quote `"`

`dollar` Expands to '`$`'.

`lf` Expands to a line-feed character (Unix end of line).

`n` Expands to a newline character.

`quote` Expands to a single quote `'`.

`spc` Expands simply to a single space. This can be used to place spaces in filenames etc.

`tab` Expands to a single tab character.

You can use variables in the following places:

- In any directory name. The `$(binserver)` variable is not always appropriate in this context. For instance

```
links:

  osf::
      /$(site)/${host}/directory -> somefile
```

- In any quoted string. (See `shellcommands` in the Reference manual).

```
shellcommands:

  any::

   "/bin/echo $(timezone) | /bin/mail $(sysadm)"
   '/bin/echo "double quotes!"'
```

The latter possibility enables cfengine's variables to be passed on to user-defined scripts.

- To define the values of options passed to various actions, in the form *option*`=$(variable)`.

Variables can be defined differently under different classes by preceding the definition with a class name. For example:

```
control:

   sun4::  my_macro = ( User_string_1 )
   irix::  my_macro = ( User_string_2 )
```

Here the value assigned to `$(my_macro)` depends on which of the classes evaluates to true. This feature can be used to good effect to define the mail address of a suitable system administrator for different groups of host.

```
control:

 physics::   sysadm = ( mark,fred )
 chemistry:: sysadm = ( localsys@domain )
```

Note, incidentally, that the '`-a`' option can be used to print out the mail address of the system administrator for any wrapper scripts.

3.3 Undefined variables

Note that macro-variables which are undefined are not expanded as of version 1.6 of cfengine. In earlier versions, undefined variables would be replaced by an empty string, as in Perl. In versions 1.6.x and later, the variable string remains un-substituted, if the varaiable does not exist. For instance,

```
control:

  actionsequence = ( shellcommands )

  myvar = ( "test string " )

shellcommands:

 "/bin/echo $(myvar) $(myvar2)"
```

results in:

```
cfengine:host: Executing script /bin/echo test string  $(myvar2)
cfengine:host:/bin/echo test : sh: syntax error at line 1: `(' unexpected
cfengine:host: Finished script /bin/echo test string  $(myvar2)
```

This allows variables to be defined on-the-fly by modules.

3.4 Defining classes and making exceptions

Cfengine communicates with itself by passing messages in the form of classes. When a class becomes switched on or off, cfengine's program effectively becomes modified. There are several ways in which you can switch on and off classes. Learning these fully will take some time, and only then will you harness the full power of cfengine.

- Classes may be defined manually from the command line.
- Classes may be defined locally in the actionsequence in order to execute only some of the actions within a special category.
- Classes may become defined if cfengine actually needs to carry out an action to repair the system's configuration.
- Classes may be defined by user-defined plug-in modules.

Because cfagent works at a very high level, doing very many things for very few lines of code it might seem that some flexibility is lost. When we restrict certain actions to special classes it is occasionally useful to be able to switch off classes temporarily so as to cancel the special actions.

3.4.1 Command line classes

You can define classes of your own which can be switched on and off, either on the command line or from the action sequence. For example, suppose we define a class *include*. We use `addclasses` to do this.

```
addclasses = ( include othersymbols )
```

The purpose of this would be to allow certain 'excludable actions' to be defined. Actions defined by

```
any.include::
               actions
```

will normally be carried out, because we have defined `include` to be true using `addclasses`. But if cfagent is run in a restricted mode, in which `include` is set to false, we can exclude these actions.

So, by defining the symbol `include` to be false, you can exclude all of the actions which have `include` as a member. There are two ways in which this can be done, one is to negate a class globally using

```
cfagent -N include
```

This undefines the class `include` for the entire duration of the program.

3.4.2 actionsequence classes

Another way to specify actions is to use a class to select only a subset of all the actions defined in the actionsequence. You do this by adding a class name to one on the actions in action sequence by using a dot '`.`' to separate the words. In this case the symbol only evaluates to 'true' for the duration of the action to which it attached. Here is an example:

```
links.onlysome
shellcommands.othersymbols.onlysome
```

In the first case `onlysome` *is defined to be true* while this instance of `links` is executed. That means that only actions labelled with the class `onlysome` will be executed as a result of that statement. In the latter case, both `onlysome` and `othersymbols` are defined to be true for the duration of `shellcommands`.

This syntax would normally be used to omit certain time-consuming actions, such as tidying all home directories. Or perhaps to synchronize certain actions which have to happen in a certain order.

3.4.3 shellcommand classes

For more advanced uses of cfengine you might want to be able to define a class on the basis of the success or failure of a user-program, a shell command or user script. Consider the following example

```
groups:

   have_cc = ( "/bin/test -f /usr/ucb/cc"
               "/bin/test -f /local/gnu/cc"  )
```

Note that as of version 1.4.0 of cfengine, you may use the word `classes` as an alias for `groups`. Whenever cfagent meets an object in a class list or variable, which is surrounded by either single, double quotes or reversed quotes, it attempts to execute the string as a command passed to the Bourne shell. If the resulting command has return code zero (proper exit) then the class on the left hand side of the assignment (in this case 'have_cc') will be true. If the command returns any other value (an error number) the result is false. Since groups are the logical OR of their members (it is sufficient that one of the members matches the current system), the class 'have_cc' will be defined above if either '/usr/ucb/cc' or '/local/gnu/cc' exist, or both.

3.4.4 Feedback classes

Classes may be defined as the result of actions being carried out by cfagent. For example, if a file gets copied, needs to be edited or if diskspace falls under a certain threshhold, cfagent can be made to respond by activating classes at runtime. This allows you to create dynamically responsive programs which react to the changing environment. These classes are defined as part of other statements with clauses of the form

```
define=classlist
```

Classes like these should generally be declared at the start of a program unless the `define` statements always precede the actions which use the defined classes, with `addinstallable`.

3.4.5 Writing plugin modules

If the regular mechanisms for setting classes do not produce the results you require for your configuration, you can write your own routines to concoct the classes of your dreams. Plugin modules are added to cfagent programs from within the actionsequence, (see Reference manual). They allow you to write special code for answering questions which are too complex to answer using the other mechanisms above. This allows you to control classes which will be switched on and the moment at which your module attempts to evaluate the condition of the system.

Modules must lie in a special directory defined by the variable `moduledirectory`. They must have a name of the form '`module:`*mymodule*' and they must follow a simple protocol. Cfagent will only execute a module which is owned either by root or the user who is running cfagent, if it lies in the special directory and has the special name. A plug-in module may be written in any language, it can return any output you like, but lines which begin with a '+' sign are treated as classes to be defined (like '-D'), while lines which begin with a '-' sign are treated as classes to be undefined (like '-N'). Lines starting with '=' are variables/macros to be defined. Any other lines of output are cited by cfagent, so you should normally make your module completely silent. Here is an example module written in perl. First we define the module in the cfagent program:

```
control:

  moduledirectory = ( /local/cfagent/modules )

  actionsequence = (
                   files
                   module:myplugin
                   "module:argplugin arg1 arg2"
                   copy
                   )
...
```

```
AddInstallables = ( specialclass )
```

Note that the class definitions for the module should also be defined in as `AddInstallables`, if this is more convenient. NOTE: you *must* declare the classes before using them in the cfagent configuration, or else those actions will be ignored. Next we write the plugin itself.

```
#!/usr/bin/perl
#
# module:myplugin
#

  # lots of computation....

if (special-condition)
   {
   print "+specialclass";
   }
```

Modules inherit the environment variables from cfagent and accept arguments, just as a regular shellcommand does.

```
#!/bin/sh
#
# module:myplugin
#

/bin/echo $*
```

Cfagent defines the classes as an environment variable so that programs have access to these. E.g. try the following module:

```
#!/usr/bin/perl

print "Decoding $ENV{CFALLCLASSES}\n";

@allclasses = split (":","$ENV{CFALLCLASSES}");

while ($c=shift(@allclasses))
  {
  $classes{$c} = 1;
  print "$c is set\n";
  }
```

Modules can define macros in cfagent by outputting strings of the form

```
=variablename=value
```

3.5 The generic class any

The generic wildcard `any` may be used to stand for any class. Thus instead of assigning actions for the class `sun4` only you might define actions for any architecture by specifying:

```
any::
      actions
```

If you don't specify any class at all then cfengine assumes a default value of `any` for the class.

3.6 Debugging tips

A useful trick when debugging is to eliminate unwanted actions by changing their class name. Since cfengine assumes that any class it does not understand is the name of some host, it will simply ignore entries it does not recognize. For example:

```
myclass::
```

can be changed to

```
Xmyclass::
```

Since `Xmyclass` no longer matches any defined classes, and is not the name of any host it will simply be ignored. The `-N` option can also be used to the same effect. (see Reference manual).

3.7 Access control

It is sometimes convenient to be able to restrict the access of a program to a handful of users. This can be done by adding an access list to the `control:` section of your program. For example,

```
control:
    ...
    access = ( mark root )
```

would cause cfengine to refuse to run the program for any other users except mark and root. Such a restriction would be useful, for instance, if you intended to make set-user-id scripts but only wished certain users to be able to run them. If the access list is absent, all users can execute the program.

Note: if you are running cfagent via the `cfrun` program then cfagent is always started with the same user identity as the cfservd process on the remote host. Normally this is the root user identity. This means that the access keyword will have no effect on the use of the command `cfrun`.

3.8 Wildcards in directory names

In the two actions `files` and `tidy` you define directory names at which file checking or tidying searches should start. One economical feature is that you can define a whole group of directories at which identical searches should start in one fell swoop by making use of *wildcards*. For example, the directory names

```
/usr/*/*
/bla/*/ab?/bla
```

represent all of the *directories* (and only directories) which match the above wildcard strings. Cfagent opens each matching directory and iterates the action over all directories which match.

The symbol '`?`' matches any single character, whereas '`*`' matches any number of characters, in accordance with shell file-substitution wildcards.

When this notation is used in directory names, it always defines the starting point for a search. It does not tell the command how to search, only where to begin. The `pattern` directive in `tidy` can be used to specify patterns when tidying files and under `files` all files are considered, (see Reference manual),

3.9 Recursive file sweeps/directory traversals

File sweeps are searches through a directory tree in which many files are examined and considered for processing in some way. There are many instances where one uses cfagent to perform a file sweep.

- As part of a `files` action, for checking access rights and ownership of files.
- As part of a `tidy` action, for checking files for deletion.
- As part of a `copy` action, while recursively checking whether to copy a file tree.
- As part of an `editfiles` action, while recursively checking whether to edit the files in a tree of files.

The problem with file sweeps is that they can be too sweeping! Often you are not interested in examining every single file in a file tree. You might wish to perform a search

- excluding certain named directories and their subdirectories with `ignore`.
- excluding certain files and directories matching a specific pattern.
- including only a subset of files matching specific patterns.
- filter out very specific file types.

The tidy action is slightly different in this respect, since it already always expects to match a specific pattern. One is generally not interested in a search which deletes everything except for a named pattern: this would be too dangerous. For this reason, the syntax of `tidy` does not allow ignore, include and exclude. It is documented in the section on tidying, (see Reference manual).

Items declared under the global `ignore` section affect files, copy, links and tidy. For file sweeps within files, copy and links, you may provide private ignore lists using `ignore=`. The difference between exclude and ignore is that ignore can deal with absolute directories. It prunes directories, while exclude only looks at the files within directories.

For file sweeps within `files` and `copy` you can specify specific search parameters using the keywords `include=` and `exclude=` and as of version 1.6.x `filter=`. For example,

```
files:

   /usr/local/bin m=0755 exclude=*.ps action=fixall
```

In this example cfagent searches the entire file tree (omitting any directories listed in the ignore-list and omitting any files ending in the extension '`.ps`'), (see Reference manual).

Specifying the `include=` keyword is slightly different since it automatically restricts the search to only named patterns (using `*` and `?` wildcards), whenever you have one or more instances of it. If you include patterns in this way, cfagent ignores any files which do not match the given patterns. It also ignores any patterns which you have specified in the global ignore-list as well as patterns excluded with `exclude=`*`pattern`*. In other words, exclusions always override inclusions.

If you exclude a pattern or a directory and wish to treat it in some special way, you need to code an explicit check for that pattern as a separate entity. For example, to handle the exluded '`.ps`' files above, you would need to code something like this:

```
files:

   /usr/local/bin m=0644 include=*.ps action=fixall
```

Note: don't be tempted to enclose your wildcards in quotes. The quotes will be treated literally and the pattern might not match the way you would expect.

For `editfiles` the syntax is somewhat different. Here one needs to add lines to the edit stanza:

```
editfiles:

 { /tmp/testdir

 Include .*
 Exclude bla.*
 Ignore "."
 Ignore ".."
 Recurse 6

 ReplaceAll "search" With "replace"
 }
```

3.10 Security in Recursive file sweeps

Recursively descending into directories and performing a globally 'destructive' change is an inherently risky thing to do, unless you are certain of the directory structure.

Suppose, for instance, that a user with write access to the filesystem added a symbolic link to '`/etc/passwd`', and we were doing a recursive deletion. Suddenly, cfengine becomes a destructive weapon. The default behaviour is that cfengine does not follow symbolic links in recursive descents, for this reason. The option `travlinks` can be set to true, in order to change this. However, in general, you should never change this option, especially if untrusted users have access to parts of the filesystem, e.g. if you clear '`/tmp`' recursively.

Cfagent checks for link race attacks, in which users try to swap a directory for a link, in between system calls, to trick cfagent into believing that a link is a directory, as of version 2.0.3 (and 1.6.4).

Note that, even if `travlinks` is set to true, cfagent will not follow symbolic links that are not owned by the agent user ID; this is to minimize the possibilty of link race attacks, in which users with write access could divert the agent to another part of the filesystem.

3.11 Log files written by cfagent

Cfagent keeps two kinds of log-file privately and it allows you to log its activity to syslog. Syslog logging may be switched on with the `Syslog` variable, (see Reference manual).

The first log cfagent keeps is for every user (every subdirectory of a home directory filesystem). A file `~/.cfengine.rm` keeps a list of all the files which were deleted during the last pass of the `tidy` function. This is useful for users who want to know files have been removed without their blessing. This helps to identify what is happening on the system in case of accidents.

Another file is built when cfagent searches through file trees in the `files` action. This is a list of all programs which are setuid root, or setgid root. Since such files are a potential security risk, cfagent always prints a warning when it encounters a new one (one which is not already in its list). This allows the system administrator to keep a watchful eye over new programs which appear and give users root access. The cfengine log is called `/var/cfengine/cfengine.log`. The file is not readable for general users.

3.12 Quoted strings

In several cfengine commands, you use quoted strings to define a quantity of text which may contain spaces. For example

```
control:

  macro = ( "mycommand" )

editfiles:

  { $(HOME)/myfile

   AppendIfNoSuchLine 'This text contains space'
  }
```

In each case you may use any one of the three types of quote marks in order to delimit strings,

```
' or " or `
```

If you choose, say `"`, then you may not use this symbol within the string itself. The same goes for the other types of string delimiters. Unlike the shell, cfengine treats these three delimiters in precisely the same way. There is no difference between them. If you need to quote a quoted string, then you should choose a delimiter which does not conflict with the substring before version 2.0.7. From version 2.0.7, you can escape quotes.

```
qstring = ( "One string\"with substring\" escaped" )
```

Note that you can use special variables for certain symbols in a string See Section 3.2 [Variable substitution], page 24.

3.13 Regular expressions

Regular expressions can be used in cfagent in connection with `editfiles` and `processes` to search for lines matching certain expressions. A regular expression is a generalized wildcard. In cfagent wildcards, you can use the characters '*' and '?' to match any character or number of characters. Regular expressions are more complicated than wildcards, but have far more flexibility.

NOTE: the special characters '`*`' and '`?`' used in wildcards do not have the same meanings as regular expressions!.

Some regular expressions match only a single string. For example, every string which contains no special characters is a regular expression which matches only a string identical to itself. Thus the regular expression ‘cfengine’ would match only the string "cfengine", not "Cfengine" or "cfengin" etc. Other regular expressions could match more general strings. For instance, the regular expression ‘c*’ matches any number of c’s (including none). Thus this expression would match the empty string, "c", "cccc", "ccccccccc", but not "cccx".

Here is a list of regular expression special characters and operators.

‘\’ The backslash character normally has a special purpose: either to introduce a special command, or to tell the expression interpreter that the next character is not to be treated as a special character. The backslash character stands for itself only when protected by square brackets [\] or quoted with a backslash itself ‘\\’.

‘\b’ Matches word boundary operator.

‘\B’ Match within a word (operator).

‘\<’ Match beginning of word.

‘\>’ Match end of word.

‘\w’ Match a character which can be part of a word.

‘\W’ Match a character which cannot be part of a word.

‘*any character*’
Matches itself.

‘.’ Matches any character

‘*’ Match zero or more instances of the previous object. e.g. ‘c*’. If no object precedes it, it represents a literal asterisk.

‘+’ Match one or more instances of the preceding object.

‘?’ Match zero or one instance of the preceding object.

‘{ }’ Number of matches operator. ‘{5}’ would match exactly 5 instances of the previous object. ‘{6,}’ would match at least 6 instances of the previous object. ‘{7,12}’ would match at least 7 instances of, but no more than 12 instances of the preceding object. Clearly the first number must be less than the second to make a valid search expression.

‘|’ The logical OR operator, OR’s any two regular expressions.

‘[*list*]’ Defines a list of characters which are to be considered as a single object (ORed). e.g. ‘[a-z]’ matches any character in the range a to z, ‘abcd’ matches either a, b, c or d. Most characters are ordinary inside a list, but there are some exceptions: ‘]’ ends the list unless it is the first item, ‘\’ quotes the next character, ‘[:’ and ‘:]’ define a character class operator (see below), and ‘-’ represents a range of characters unless it is the first or last character in the list.

‘[^*list*]’ Defines a list of characters which are NOT to be matched. i.e. match any character except those in the list.

‘‘[:*class*:]’’
Defines a class of characters, using the ctype-library.

`alnum`	Alpha numeric character
`alpha`	An alphabetic character
`blank`	A space or a TAB
`cntrl`	A control character.
`digit`	0-9
`graph`	same as print, without space
`lower`	a lower case letter
`print`	printable characters (non control characters)
`punct`	neither control nor alphanumeric symbols
`space`	space, carriage return, line-feed, vertical tab and form-feed.
`upper`	upper case letter
`xdigit`	a hexadecimal digit 0-9, a-f

‘‘`( )`’’ Groups together any number of operators.

‘`\`*`digit`*’ Back-reference operator (refer to the GNU regex documentation).

‘`^`’ Match start of a line.

‘`$`’ Match the end of a line.

Here is a few examples. Remember that some commands look for a regular expression match of part of a string, while others require a match of the entire string (see Reference manual).

```
^#        match string beginning with the # symbol
^[^#]      match string not beginning with the # symbol
^[A-Z].+  match a string beginning with an uppercase letter
          followed by at least one other character
```

3.14 Iterating over lists

Shell list variables are normally defined by joining together a list of directories using a concatenation character such as ‘`:`’. A typical example of this is the PATH variable:

```
PATH=/usr/bin:/usr/local/bin:/usr/sbin
```

It is convenient to be able to use such variables to force cfagent to iterative over a list. This gives us a compact way of writing repeated operations and it allows a simple method of communication with the shell environment. For security reasons, iteration is supported only in the following contexts:

- in the ‘to’ field of a multiple link action,
- in the ‘from’ field of a copy action,
- in the directory field of a tidy action,

- in the directory field of the files action,
- in the ignore action,
- in a shell command.

This typically allows communication with PATH-like environment variables in the shell.

In these contexts, any variable which has the form of a list joined together by colons will be iterated over at compilation time. Note that you can change the value of the list separator using the `Split` variable in the control section of the program (see Reference manual).

For example, to link all of the binary files in the PATH environment variable to a single directory, tidying dead links in the process, you would write

```
control:

  actionsequence = ( links tidy )

links:

  /allbin +> $(PATH)

tidy:

  # Hopefully no-match matches nothing

  /allbin pattern=no-match age=0 links=tidy
```

no-match is not a reserved word in cfengine, this is just a string you do not expect to match any file.

Alternatively, you might want to define an internal list using a space as a separator:

```
control:

   Split = ( " " )

   mylist = ( "mark ricky bad-dude" )

tidy:

   /mnt/home1/$(mylist) pattern=*.cfsaved age=1
```

This example iterates the tidy action over the directories '`/mnt/home1/mark`', '`/mnt/home1/ricky`' and '`/mnt/home1/bad-dude`'.

The number of list variables in any path or filename should normally be restricted to one or two, since the haphazard combination of two lists will seldom lead to any meaningful pattern. The only obvious exception is perhaps to iterate over a common set of child-directories like '`bin`', '`lib`' etc in several different package directories.

4 Designing a global system configuration

This chapter is about building strategies for putting together a site configuration for your entire network.

4.1 General considerations

In order to use any system administration tool successfully, you have to make peace with your system by deciding exactly what you expect and what you are willing to do to achieve the results. You need to decide what you will consider to be acceptable and what is to be considered completely untenable. You need to make these decisions because otherwise you will only be confused later when things don't go the way you expected.

Experience shows that the most successful policies for automation involve keeping everything as simple as possible. The more uniform or alike your machines are, the easier they are to run and the happier users are. Sometimes people claim that they need such great flexibility that all their machines should be different. This belief tends to be inversely proportional to the number of machines they run and generally only applies to very special development environments! Usually you will only need one or two machines to be special and most can be made very similar.

Site configuration is about sharing and controlling resources. The resources include disks (filespace), files, data, programs, passwords and physical machines. Before planning your sitewide configuration you should spend some time deciding how you would like things to work.

In the remaining parts of this chapter, you will find some hints and tips about how to proceed, but remember that when push comes to shove, you must make your own choices.

4.2 Using netgroups

If you use the network information service (NIS) on your local network then you may already have defined *netgroups* consisting of lists of hosts which belong to specific owners at your site. If you have, then you can use these groups within cfengine. This means that you can use the same groups in the `/etc/exports` file as you use to define the mount groups and classes.

A netgroup is a list of hostnames or user names which are registered in the network information service (NIS) database under a specific name. In our case we shall only be interested in lists of hostnames.

To make a netgroup you need to define a list in the file `/etc/netgroup` on your NIS server. If you are not the NIS administrator, you will have to ask to have a netgroup installed. The form of a netgroup list of hosts is:

```
mylist-name      (host1,,) (host2,,) (host3,,) (host4,,)

norway-sun4-host (saga,,) (tor,,) (odin,,)
foes-linux-hosts (borg,,)
```

Each list item has three entries, but only the first is relevant for a host list. See the manual pages on netgroups for a full explanation of the meaning of these fields.

The usefulness of netgroups is that they can be used to stand for a list of hostnames in system files like '`/etc/exports`'. This compresses the amount of text in this file from a long list to a single name. It also means that if you use the same list of hosts from a netgroup inside cfengine when defining groups and classes, you can be sure that you are always using the same list. In particular it means that you don't have to update multiple copies of a list of hosts.

The netgroups can now be used in cfengine programs by using the `+` or `@+` symbols in the `groups` section. (see Reference manual).

4.3 Files and links

File and link management takes several forms. Actions are divided into three categories called `files`, `tidy` and `links`. The first of these is used to check the existence of, the ownership and permissions of files. The second concerns the systematic deletion of garbage files. The third is a link manager which tests, makes and destroys links. The monitoring of file access bits and ownership can be set up for individual files and for directory trees, with controlled recursion. Files which do not meet the specified criteria can be 'fixed' —i.e. automatically set to the correct permissions, or can simply be brought to the attention of the system administrator by a warning. The syntax of such a command is as follows:

```
files:

  class::

    /path mode=mode owner=owner group=group

         recurse=no-of-levels action=action
```

The directory or file name is the point at which cfagent begins looking for files. From this point the search for files proceeds recursively into subdirectories with a maximum limit set by the `recurse` directive, and various options for dealing with symbolic links and device boundaries. The mode-string defines the allowed file-mode (by analogy with '`chmod`') and the owner and group may specify lists of acceptable user-ids and group-ids. The action taken in response to a file which does not meet acceptable criteria is specified in the action directive. It includes warning about or directly fixing all files, or plain files or directories only. Safe defaults exist for these directives so that in practice they may be treated as options.

For example,

```
files:

  any::
       /usr/*/bin mode=a+rx,o-w own=root r=inf act=fixall
```

which (in abbreviated form) would check recursively all files and directories starting from directories matching the wildcard (e.g. '`/usr/local/bin`', '`/usr/ucb/bin`'). By default, `fixall` causes the permissions and ownership of the files to be fixed without further warning.

One problem with symbolic links is that the files they point to can get deleted leaving a 'hanging pointer'. Since cfagent can make many hundreds of links without any effort, there is the danger that, in time, the system could become full of links which don't point anywhere. To combat this problem, you can set the option `links=tidy` in the files section. If this is set, cfagent will remove any symbolic links which do not point to existing files (see Reference manual).

The creation of symbolic links is illustrated in figure 1 and the checking algorithm was discussed in section 2. In addition to the creation of single links, one may also specify the creation of multiple links with a single command. The command

```
links:

  binaryhost::

    /local/elm/bin +> /local/bin
```

links all of the files in '`/local/elm/bin`' to corresponding files in '`/local/bin`'. This provides, amongst other things, one simple way of installing software packages in regular 'bin' directories without controlling users' PATH variable. A further facility makes use of cfagent's knowledge of available (mounted) binary resources to search for matches to specific links. Readers are referred to the full documentation concerning this feature.

The need to tidy junk files has become increasingly evident during the history of cfengine. Files build up quickly in areas like '`/tmp`', '`/var/tmp`'. Many users use these areas for receiving large ftp-files so that their disk usage will not be noticed! To give another example, just in the last few months the arrival of netscape World Wide Web client, with its caching facilities, has flooded hard-disks at Oslo with hundreds of megabytes of WWW files. In addition the regular appearance of '`core`' files[1] and compilation by-products ('`.o`' files and '`.log`' files etc.) fills disks with large files which many users do not understand. The problem is easily remedied by a few lines in the cfagent configuration. Files can be deleted if they have not been accessed for n-days. Recursive searches are both possible and highly practical here. In following example:

```
tidy:

  AllHomeServers::

    home                  pattern=core       r=inf age=0
    home/.wastebasket     pattern=*          r=inf age=14
    home/.netscape-cache pattern=cache????* r=inf age=2
    home/.MCOM-cache      pattern=cache????* r=inf age=2
    home/.netscape        pattern=cache????*  r=inf age=2
```

all hosts in the group '`AllHomeServers`' are instructed to iterate over all users' home directories (using the wildcard `home`) and look for files matching special patterns. Cfagent tests the *access time* of files and deletes only files older than the specified limits. Hence all core files, in this example, are deleted immediately, whereas files in the subdirectory '`.wastebasket`' are deleted only after they have lain there untouched for 14 days, and so on.

As a system administrator you should, of course, exercise great caution when making rules which can delete users' files. A single slip of the hand can result in a rule which will irretrievably delete files.

[1] On some systems, core dumps cannot be switched off!

When making a 'tidy' strategy you should probably coordinate with your backup policy. You should not delete files until after you have taken a backup, so that — if the worst should happen — you are covered against possible accidents.

Cfagent helps to some extent to keep track of what files it deletes. When tidying users' home directories it creates a log file of all files which were deleted on the last tidy operation. This log is called `~/.cfengine.rm`.

You might consider tidying certain files only once a week, in which case a command such as

```
tidy:

   AllHomeServers.Sunday::

       files to tidy
```

could be useful. Nonsense files, such as 'core' files could be tidied every night.

NOTE! Be careful when telling cfagent to delete core files. If you write a wildcard like `core*`*, then you could risk deleting important system files such as* `core.h`*.*

4.4 Copying files

The administration of a system often requires the copying of files. The reason for this is usually that we would like to distribute a copy of a particular file, from some master location and ensure that all of the copies are up to date. Another use for this is to install software from one directory (perhaps on a CD ROM) to another.

Cfagent helps this process by allowing you to copy a single file or a file tree, from one directory to another, perhaps checking the permissions and owners of a file to adjust the copies in some special way. The files are checked by cfagent using one of two methods.

- A date-stamp comparison with a master file, using last-change times, can be used to tell cfagent to recopy a file from the master if the master file is newer than the copy.
- A checksum can be computed for each file and compared with one for the master file. If the contents of the copy file differs in any way from the master, the file will be re-copied.

Cfengine allows you to do the following

- Copy a single file to another file in a different location, perhaps with a new name, new permissions and a different owner.
- Copy a single file to all users on the system, changing the owner of the file for each user automatically. (This could be used to distribute and update standard setup files.)
- Recursively copy an entire file tree, omitting files which match a list of wildcard-patterns, or linking certain files instead of copying.

You can find out more about copying in the reference section.

4.5 Managing processes

Cfagent allows you to check for the existence of processes on your system, send those processes signals (such as kill) and perhaps restart those processes. Typical applications for this are sending '`cron`' and '`inetd`' the HUP signal, after editing their configuration files, or killing unwanted processes (such as user programs which hog the system at peak usage times).

You can read more about this in the reference section .

4.6 Cfengine's model for NFS-mounted filesystems

Most of the filesystems that you will want to make available across the network are going to fall into one of two categories. In cfengine parlance these are called *home directories* and *binary directories*. A home directory is a place where users' login directories are kept. This is traditionally a directory called '`/home`' or '`/users`' or some subdirectory of these. A binary directory is a place where compiled software is kept. Such files (which do not belong to the pure operating system release) are often placed in a directory called '`/usr/local`' or simply '`/local`'.

In this chapter we shall consider a scheme for using cfengine to make NFS filesystem management quite painless.

4.6.1 NFS filesystem resources

Using the Network File System (NFS) in a large workstation environment requires a bit of planning. The idea of NFS is to share files on one host with other hosts. In most cases, filesystems to be shared across the network fall into two categories: *binary* filesystems (those which contain compiled software) and *user* or *home* filesystems (which contain users' login areas).

The most simple minded way to share resources would be to mount every resource (each available NFS filesystem) onto every host. To avoid collisions, each filesystem would have to have a unique name. This is one possibility, but not a very intelligent one. As experienced users will realize, cross-mounting too many NFS filesystems is a recipe for all kinds of trouble.

Cfengine offers a simple model which can help you pick out only the resources you need from the list of NFS filesystems. It will then mount them automatically and edit the appropriate filesystem tables. It does this by defining classes of hosts. For instance — you really don't need to mount a binary filesystem for an `ultrix` system onto an `HPUX` system. There would be no point — binary resources are *architecture* or *hard-class dependent*. But home directories are architecture independent.

Cfengine lets you to define a list of allowed servers for various hosts so that only filesystems from the servers will be considered for mounting!

4.6.2 Unique filesystem mountpoints

The first step towards treating NFS filesystems as network resources is to invent a naming scheme so that every filesystem has a unique name on which it can be mounted. If we don't sort this out now, we could find two or more hosts with a filesystem called `/usr/local`, both of which we might like to mount since they contain different software.

A simple but extremely useful naming scheme is the following.[2] If you don't like this scheme you can invent your own, but the remainder of the text will encourage you to use this one. If you follow this scheme, exactly as described here, you will never have any problems with mount points. We shall describe the scheme in detail below. Here are some points to digest:

- When mounting a remote filesystem on your local system, the local and remote directories should always have exactly the same name.
- The name of every filesystem mountpoint should be unique and tell you something meaningful about where it is located and what its function is.
- You can always make links from special unique names to more general names like '`/usr/local`'. If you this involves compiled software and you do this on one host, you should do it on others which are of the same type.
- It doesn't matter whether software compiles in the path names of special directories into software as long as you follow the points above.

Each filesystem is given a directory name composed of three parts:

```
/site/host/contents
```

The first directory (which only exists to create a suitable mountpoint) is the name of your local site. If you are a physics department at a university (with a separate setup) you could call this 'physics'. It could be your company name or whatever. The second piece is the name of the host to which the disk space is physically attached. The final piece is the name of the filesystem. Here are some typical examples:

```
/physics/einstein/local    # /usr/local for einstein@physics
/physics/newton/u1         # user partition 1 for newton@physics
```

On the machines which are home to the 'local' partition, it is better to make a link to `/usr/local` than call the filesystem `/usr/local` directly. This is because it makes the procedure of organizing the entire network much clearer.

It is worth noting that, when you ask cfagent to mount such a resource, it will automatically make the mount directory and can easily be asked to make a link to `/usr/local`, so this small amount of extra work is really no work at all.

The whole naming convention is compactly summarized by defining a mount point variable, `mountpattern`. With the present scheme, this can be defined as

```
mountpattern = ( /$(site)/$(host) )
```

so that it evaluates to the name of the host executing the file regardless of who that may be. This variable is used together with the `homepattern` pattern variable, which is used to distinguish between home directories and binary resources. (See `homepattern` in the reference section). You can think of this as being part of the naming convention. In this text, we use the convention `u1 u2 u3...` for home disks. You could equally well use `home1 home2...` etc. As long as the name is unique, it doesn't matter.

The full list of named resources should now be listed in the `mountables` list, which is simply a list of all the resources available for mounting on the network.

---

[2] This unique naming scheme was suggested to me originally by Knut Borge at USIT of the University of Oslo.

4.6.3 How does it work?

Once you have defined your unique names, how does cfagent know what to mount? The idea is now to define a list of servers for each class of hosts.

Suppose we make a `binserver` declaration:

```
binservers:

  mygroup.sun4::

     einstein
     newton
```

This would tell cfagent that it should mount all binary resources from hosts `einstein` or `newton` onto any host of type `sun4` in the group `mygroup`. Every filesystem which is listed in `mountables` and is not a home directory will be mounted.

Home directories and binary resources are kept separate automatically by cfagent, because a home directory is one whose contents-name matches the `homepattern` pattern variable. See Section 4.6.2 [Unique filesystem mountpoints], page 43.

A `homeserver` declaration:

```
homeservers:

  mygroup::

     einstein
     newton
     schwinger
     feynman
```

would correspondingly mean mount all the home directory resources on the hosts in the list on all hosts in the group `mygroup`. Clearly it is unnecessary to distinguish between the architecture platform types of the actual servers for user directories.

In each case, cfagent will mount filesystems, make the appropriate directories for the mount point and edit the filesystem table.

4.6.4 Special variables

Once you have mounted a resource on a unique directory, you have access to all of the relevant filesystems on your network — but you really wanted the 'local' filesystem to be mounted on `/usr/local`. All you need do now is to make a link:

```
links:

  any::

      /usr/local  -> /$(site)/$(binserver)/local
```

The meaning of this is that, on any host, the directory `/usr/local` should be a link to the 'nearest' binary server's 'local' resource. The `$(binserver)` variable can in principle expand to any binary server in the list. In practice, cfagent goes through the list in order and picks the first filesystem resource which matches.

Could this lead to a collision? Suppose we are on the host 'einstein' and we execute the above command. The host 'einstein' has a filesystem `/physics/einstein/local` on its local disk — it is in fact the binary server for the network, so it certainly doesn't need to mount any NFS filesystems. But this is no problem because cfagent automatically treats `$(host)` as the highest priority binary server for any host. That means that if you have a local filesystem, it will always have priority.

In contrast, if the host 'schwinger' ran the command above, it would find no local filesystem called `/physics/schwinger/local`, so it would go along the list of defined binary servers, find 'einstein' and try again. It will succeed in finding 'einstein' *provided all the binary servers were mounted before the link command is executed*. This means that you should structure the `actionsequence` so that all filesystems are mounted before any links are made.

With a little practice, the cfengine model can lead to an enormous simplification of the issue of NFS-mountable resources.

NOTE: cfengine does not try to export filesystems, only mount already exported filesystems. If you want to automate this procedure also, you can use the `editfiles` facility to add entries to '`/etc/exports`' (see `editfiles` in the Reference manual). In practice this is very difficult to do and perhaps not desirable.

4.6.5 Example programs for mounting resources

Let's write a very simple configuration for a network with only one server called hal, where all the hosts are of the same operating system type. In such an example we can avoid using classes altogether.

```
control:

  site   = ( univ )
  domain = ( univ.edu )

  actionsequence =
     (
     mountall
     mountinfo
     addmounts
     mountall
     links
     )

  mountpattern = ( /univ )
  homepattern = ( home? )

binservers:

   hal

homeservers:

   hal

mailserver:

   hal:/var/spool/mail
```

```
mountables:

   hal:/univ/home1
   hal:/univ/home2
   hal:/univ/local

links:

   /usr/local -> /univ/local
```

In this example, we have only one type of host so the configuration is the same for each of them: no class references are required. If we look through the action sequence we see that the program first mounts all the filesystems which are already defined on each host. It does this to be sure that everything which is already set up to be mounted is mounted. Let's assume that there are no problems with this.

The next thing that happens is that `mountinfo` builds a list of the filesystems which each host has successfully mounted. Then by calling `addmounts` we ask cfagent to check whether the host is missing any filesystems. What happens is that cfagent first looks to see what servers are defined for each host. In this case all hosts on the network have only one server: hal. Hal is defined as a server for both binary data and 'home' data — i.e. users' home directories. The list `mountables` tells cfagent what filesystems are available over the network for the server hal. There are three filesystems which can be mounted, called '`/univ/home1`', '`/univ/home2`' and '`/univ/local`'. Cfagent checks to see whether each of these filesystems is mounted and, if not, it builds the necessary directories, edits the necessary files and mounts the filesystems.

Finally we come to `links` in the action sequence. This tells cfagent to look at the defined links. There is one link defined: a link from '`/usr/local`' to the mounted filesystem '`/univ/local`'. Cfagent checks and tries to make the link if necessary. If all goes well, each host on the network should now have at least three filesystems mounted and a link from '`/usr/local`' to '`/univ/local`'.

Here is another simple example program for checking and automatically mounting an NFS based `/usr/local` and all home directories onto all hosts on a small network. Here we have several servers and must therefore use some classes.

```
#
#  Mounts
#


control:

   site      = ( mysite )
   domain    = ( mysite.country )
   sysadm    = ( mark )
   netmask   = ( 255.255.255.0 )

   actionsequence =
      (
      mountall
      mountinfo
      addmounts
```

```
      mountall
      links
      )

   mountpattern = ( /$(site)/$(host) )
   homepattern   = ( u? )               # u1 u2 u3 etc..

groups:

   MyGroup =
      (
      host1
      host2
      binserver1
      binserver2
      )

######################################################################

homeservers:

   MyGroup:: host1


binservers:

   MyGroup.sun4::   server1
   MyGroup.ultrix:: server2

mailserver:

   host1:/usr/spool/mail

mountables:

   host1:/mysite/host1/u1
   host1:/mysite/host1/u2
   server1:/mysite/server1/local
   server2:/mysite/server2/local


##########################################################################

links:

      /usr/local  -> /${site}/${binserver}/local
```

Let's suppose we run this program on host2 which is an ultrix machine. This host belongs to the class `mygroup` and the hard-class `ultrix`. This tells us that its homeserver is host1, its binary server is server2 and its mailserver is host1. Moreover, since the homepattern matches any filesystem ending in u-something, it recognizes the two home directories in the mountables list — and therefore the two binary directories also.

The action sequence starts by mounting all of the filesystems currently in the filesystem table '`/etc/fstab`'. It then scans the list of mounted filesystems to find out what is actually mounted. Since the homeserver is host1, we know that our host has to mount all home-filesystems from this server, so it checks for 'host1:/mysite/host1/u1' and 'host1:/mysite/host1/u2'. If they are not present they are added to '/etc/fstab'[3]. Next, we know that the binary server is server1, so we should check for 'server1:/mysite/server1/local'. The mail server is also checked for and added if necessary. Cfagent then tries to mount all filesystems once again, so that the new filesystems should be added.

Note that, in the process of adding the filesystems to '/etc/fstab', cfagent creates the directories up to and including the point at which the filesystems should be mounted. If something prevents this — if we try to mount on top of a plain file for instance — then this will result in an error.

Finally, we reach the link section and we try to expand the variables. `$(site)` expands to 'mysite'. `$(binserver)` expands first to the hostname (host2), but '/mysite/host2/local' does not exist, so it then goes to the binserver list, which substitutes server1 for the value of `$(binserver)`. Since '/mysite/server1/local' does exist and is now mounted, cfagent makes a link to this directory from '/usr/local'. The script is then completed.

If the script is run again, everything should now be in place so nothing happens. If for some reason it failed the first time, it will fail again. At any rate it will either do the job once and for all or signal an error which must be corrected by human intervention[4].

4.7 Using the automounter

The automounter is a daemon based service which replaces static mounting of NFS filesystems with a dynamical model. When the automounter is running, filesystems are mounted only when a user tries to access a file which resides on one of those filesystem. After a given period (usually five minutes) any filesystem which has not been accessed is unmounted. The advantage of this scenario is that hanging servers do not affect the behaviour of hosts which mount their filesystems, unless a specific file is being accessed. In both cases, filesystems must be exported in order to be mountable.

It is not the purpose of this section to explain the use of the automounter in detail, only to offer hints as to how cfengine can be used to simplify and rationalize automount configuration for the already initiated. Let us begin by comparing the behaviour of the automounter with the cfengine model for mounted filesystems.

The automounter is designed to be used together with a global configuration file, distributed by NIS (the network information service). As such, all hosts read the same configuration file. This makes it appear as though all hosts end up mounting every filesystem in the automount configuration database, but this is not so in practice because filesystems are only mounted if required. Thus a system which does not require a filesystem will not attempt to mount it. Moreover, the existence of a global configuration file does not affect which hosts have the right to mount certain filesystems (which is specified by exports or share on the relevant server), thus a request to mount a non-exported filesystem will result

[3] Note: if the filesystem was in the fstab but not actually mounted a warning is issued telling you that the filesystem was probably not exported correctly on host1.

[4] One possibility is that an NFS filesystem cannot be mounted because the host serving the filesystem is out of service. If this is the case then a subsequent re-run when the server resumes normal service will succeed.

in an access denial. The automounter is configured locally on each host in files named '`/etc/auto_master`', '`auto_direct`' etc.

In the cfengine static mounting scheme, you define a list of binary and home servers. The filesystem table is modified on the basis of these decisions, and filesystems are only added if cfagent deems it appropriate to mount them on a given host. The idea here is to minimize the number of filesystems mounted to those which are known to be required. Again the issue of access permissions must be arranged separately. These filesystems are placed directly in '`/etc/fstab`', or the equivalent for your system.

From cfengine, you can use the automounter instead of the static mount model by

- omitting `addmounts`, `mountinfo`, `mountall` from the actionsequence, in the control part of your cfengine program,
- using `editfiles` to edit the relevant configuration files such as '`/etc/auto_master`', or '`auto_direct`' etc,
- using the `AutomountDirectResources` command in editfiles to dump the list of cfengine class-based list of mountables into a file of your choice in the correct format for autmount's direct maps,
- using `processes` to restart the automounter (send the hangup signal `hup`), or perhaps stop and restart the daemon by sending the `term` signal (you should never send the `kill` signal).
- using the multiple link facilities to link in indirect mounted filesystems as required, and `files` or `tidy` to clean up stale links afterwards,
- perhaps using `copy` to distribute basic automount configuration files to multiple systems.

The automounter was created to solve certain problems which cfengine now solves (in the author's opinion) better. For example, the use of the 'hosts' map in the automounter mounts filesystems like '`/usr/local`' on different (uniquely named) mountpoints for each host in order to avoid name space collisions. Using cfengine and a unique naming scheme, you can achieve the same thing more cleanly, without all of the gratuitous linking and unlinking which the automounter performs by itself. Moreover, the idea of a unique namespace is better practice and more in keeping with new global filesystem ideas such as AFS and DFS. The only advantage of the automounter is that one avoids the annoying error messages from hung servers about "NFS server not responding". In that respect, it seems sensible to use only direct mounts and a unique name space.

Some systems advocate grouping all users' login (home) directories under a common directory called '`/home`' or '`users`'. The automounter goes through all manner of contortions to achieve this task. If you use a unique naming scheme like the one advocated here, this is a trivial task. You simply arrange to mount or automount all user directories, such as

```
/site/host/home1
/site/host/home2
...
```

and then link them as follows:

```
/home +> /site/host/home1
/home +> /site/host/home2
...
```

Finally, you should be aware that the automounter does not like to be mixed with static mount and unmount operations. Automounted filesystems take priority over statically mounted filesystems, but the automounter can be confused by manually mounting or unmounting filesystems while it is running.

4.8 Editing Files

A very convenient characteristic of BSD and UNIX System V systems is that they are configured primarily by human-readable textfiles. This makes it easy for humans to configure the system and it also simplifies the automation of the procedure. Most configuration files are line-based text files, a fact which explains the popularity of, for example, the Perl programming language. Cfengine does not attempt to compete with Perl or its peers. Its internal editing functions operate at a higher level which are designed for transparency rather than flexibility. Fortunately most editing operations involve appending a few lines to a file, commenting out certain lines or deleting lines.

For example, some administrators consider the finger service to be a threat to security and want to disable it. This could be done as follows.

```
editfiles:

      { /etc/inetd.conf

      HashCommentLinesContaining "finger"
      }
```

Commands containing the word 'Comment' are used to 'comment out' certain lines from a text-file—i.e. render a line impotent without actually deleting it. Three types of comment were supported originally: shell style (hash) '`#`', '`%`' as used in TeX and on AIX systems, and C++-style '`//`'.

A more flexible way of commenting is also possible, using directives which first define strings which signify the start of a comment and the end of a comment. A single command can then be used to render a comment. The default values of the comment-start string is '`# `' and the default comment-end string is the empty string. For instance, to define C style comments you could write:

```
{ file

SetCommentStart "/* "
SetCommentEnd   " */"

# Comment out all lines containing printf!

CommentLinesMatching ".*printf.*"
}
```

Other applications for these editing commands include monitoring and controlling root-access to hosts by editing files such as '`.rhosts`' and setting up standard environment variables in global shell resource files— for example, to set the timezone. You can use the editing feature to update and distribute the message of the day file, or to configure sendmail, (see FAQS and Tips in the Reference manual).

An extremely powerful feature of cfagent is the ability to edit a similar file belonging to every user in the system. For example, as a system administrator, you sometimes need to ensure that users have a sensible login environment. Changes in the system might require all users to define a new environment variable, for instance. This is achieved with the `home` pseudo-wildcard. If one writes

```
{ home/.cshrc

AppendIfNoSuchLine "# Sys admin/cfengine: put next line here"
AppendIfNoSuchLine "setenv PRINTER newprinter"
}
```

then the users' files are checked one-by-one for the given lines of text, and edited if necessary.

Files are loaded into cfagent and edited in memory. They are only saved again if modifications to the file are carried out, in which case the old file is preserved by adding a suffix to the filename. When files are edited, cfagent generates a warning for the administrator's inspection so that the reason for the change can be investigated.

The behaviour of cfagent should not be confused with that of *sed* or *perl*. Some functionality is reproduced for convenience, but the specific functions have been chosen on the basis of (i) their readability and (ii) the fact that they are 'frequently-required-functions'. A typical file editing session involves the following points:

- Load file into memory.
- Is the size of the file within sensible user-definable limits? If not, file could be binary, refuse to edit.
- Check each editing command and count the number of edits made.
- If number of edits is greater than zero, rename the old file and save the edited version in its place. Inform about the edit.
- If no edits are made, do nothing, say nothing.

Equivalent one-line sed operations involve editing the same file perhaps many times to achieve the same results—without the safety checks in addition.

4.9 Disabling and the file repository

The existence of certain files can compromise the integrity of your system and you may wish to ensure that they do not exist. For example, some manufacturers sell their workstations with a '+' symbol in the file '`/etc/hosts.equiv`'. This means that anyone in your NIS domain has password free access to the system!! Since this is probably not a good idea, you will want to disable this file by renaming it, or simply deleting it.

```
disable:

   /etc/hosts.equiv
```

Other files compromise the system because they grow so large that they fill an entire disk partition. This is typically true of log files such as the System V files '`/var/adm/wtmpx`' and

'/var/lp/logs/lpsched'. Other files like */var/adm/messages* get "rotated" by the system so that they do not grow so large as to fill the disk. You can make cfagent rotate these files too, by writing

```
disable:

   Sunday::

   /var/lp/logs/lpsched  rotate=3
```

Now, when cfagent is run, it renamed the file 'lpsched' to a file called 'lpsched.1'. It also renames 'lpsched.1' as 'lpsched.2' and so on, until a maximum of 3 files are kept. After passing 3, the files 'fall off the end' and are deleted permanently. This procedure prevents any log files from growing too large. If you are not interested in keeping back-logs, then you may write `rotate=empty` and cfagent will simply empty the log file.

When ever cfagent disables a file (`disable` or `links` with the '!' operator), or saves a new file on top of an old one (`copy` or `editfiles`), it makes a backup of the original. Usually disabled files are renamed by appending the string '.cfdisabled' the filename; copied files are saved by appending the string '.cfsaved'. It is possible to switch off backup file generation in the copy feature by setting the variable `backup=false`, but a better way of managing disabled and backed-up files is to use a directory in which you collect all such files for the whole system. This directory is called the file repository and is set in the control part of the program, as follows: In the `copy` directive, the option `backup=timestamp` can be used to allow multiple backups. This is useful when using cfengine to make disk backups for users, since it allows multiple versions to co-exist.

```
control:

   repository = ( directory-name )
```

If this variable is defined, cfagent collects all backup and disabled files (except for rotated files) in this directory, using a unique pathname. You can then inspect these files in the repository and arrange to tidy the repository for old files which are no longer interesting.

4.10 Running user scripts

Above all, the aim of cfengine is to present a simple interface to system administrators. The actions which are built into the engine are aimed at solving the most pressing problems, not at solving every problem. In many cases administrators will still need to write scripts to carry out more specific tasks. These scripts can still be profitably run from cfengine. Variables and macros defined in cfengine can be passed to scripts so that scripts can make maximal advantage of the class based decisions. Also note that, since the days of the week are also classes in cfengine, it is straightforward to run weekly scripts from the cfengine environment (assuming that the configuration program is executed daily). An obvious use for this is to update databases, like the fast-find database one day of the week, or to run quota checks on disks.

```
shellcommands:

  myhost.Sunday::
```

```
        "/usr/bin/find/updatedb"
```

Cfengine scripts can be passed variables using normal variable substitution:

```
control:

   cfbin     = ( /var/cfengine/bin )
   backupdir = ( /iu/dax/backup )

shellcommands:

  "$(cfbin)/cfbackup -p -f $(backupdir) -s /iu/nexus/u1"
```

If you need to write a particularly complex script to expand cfagent's capabilities, it might be useful to have full access to the defined classes. You can do this in one of two ways:

- Pass the variable `$(allclasses)` to the script. This contains a list of all classes in the form of a string

  ```
  CFALLCLASSES=class1:class2:...
  ```

  This variable always contains an up to date list of the defined classes.
- Use the command line option '`-u`' or '`--use-env`'. When this is defined, cfagent defines an internal environment variable called '`CFALLCLASSES`' which contains the same list as above. Unfortunately, System V boxes don't seem to like having to update an environment variable continuously and tend to dump core, so this is not the default behaviour!

4.11 Compressing old log files

In the previous two sections we have looked at how to rotate old log files and how to execute shell commands. If you keep a lot of old log files around on your system, you might want to compress them so that they don't take up so much space. You can do this with a shell command. The example below looks for files matching a shell wildcard. Names of the form '`file.1`', '`file.2`'...'`file.10`' will match this wildcard and the compression program sees that they get compressed. The output is dumped to avoid spurious messages.

```
shellcommands:

  "$(gnu)/gzip /var/log/*.[0-9] /var/log/*.[0-9][0-9]  > /dev/null 2>&1"
```

Cfagent will also recognize rotated files if they have been compressed, with suffixes '`.Z`', '`.gz`', '`.rbz`' or '`.rbz`'.

4.12 Managing ACLs

Access control lists are extended file permissions. They allow you to open or close a file to a named list of users (without having to create a special group for those users). They also allow you to open or close a file for a named list of groups. Several Unix-like operating systems have had access control lists for some time; but they do not seem to have caught on.

There is a number of reasons for this dawdling in the past. The tools for setting ACLs are generally interactive and awkward to use. Because a named list of users would lead to excessive verbosity in an `ls -l` listing, one does not normally see them. There is therefore the danger that the hidden information would lead to undetected blunders in opening files to the wrong users. ACLs are also different on every vendor's filesystems and they don't work over intersystem NFS. In spite of these reservations, ACLs are a great idea. Here at Oslo College, it seems that users are continually asking how they can open a file just for the one or two persons they wish to collaborate with. They have grown used to Novell/PC networks which embraced the technology from Apollo/NCS much earlier. Previously the Unix answer to users has always been: go ask the system administrator to make a special group for you. Then do the '`chmod`' thing. And then they would say: so what's so great about this Unix then?

Addressing this lack of standardization has been the job of a POSIX draft committee. Some vendors have made their implementations in the image of this draft. Solaris 2.6 has a good implementation. In spite of this, even these systems have only awkard tools for manipulating ACLs. Not the kind of thing you want to be around much, if you have better things to do. But the incompatibility argument applies only to multiple vendor headbutting. Some institutions who share data on a global basis opt for advanced solutions to network filesystems, such as AFS and DFS. Filesystems such as DCE's DFS make extensive use of file ACLs, and they are not operating system specific. Even so, DFS provides only interactive tools for examining and setting file permissions, and this is of little use to system administrators who would rather relegate that sort of thing to a script.

The need for this kind of thing is clear. Systems which make use of ACLs for security can be brought to their knees by changing a few ACLs. Take the Apollo/Domain OS as an example. All one needs to do to kill the system is to change a few ACLs and forget what they were supposed to be. Suddenly the system is crippled, nothing works. The only solution, if you don't have a backup, is to remove all of the security. Unix has a simpler security philosophy when it comes to the operating system files, but ACLs would be a valuable addition to the security of our data.

A cfagent bare-bones file-checking program looks like this:

```
#
# Free format cfagent program
#

 control:

   ActionSequence - ( files )

 files:

   classes::

     /directory/file  mode=644
                      owner=mark,ds
                      group=users,adm
                      acl=zap
                      action=fixplain

  # ... more below
```

This program simply checks the permissions and ownership of the named file. The regular file mode, owner and group are specified straightforwardly. The new feature here is the `acl` directive. It is a deceptively simply looking animal, but it hides a wealth of complexity. The `zap` is, of course, not an access control list. Rather, cfagent uses a system of aliases to refer to ACLs, so that the clutter of the complex ACL definitions does not impair the clarity of a file command. An ACL alias is defined in a separate part of the program which looks like this:

```
# ...contd


acl:

  { zap

  method:append
  fstype:solaris
  user:rmz:rwx
  user:len:r
  }
```

As you can see, an ACL is a compound object—a bundle of information which specifies which users have which permissions. Because ACLs are *lists* the alias objects must also know whether the items are to be appended to an existing list or whether they are to replace an existing list. Also, since the permission bits, general options and programming interfaces are all different for each type of filesystem, we have to tell cfagent what the filesystem type is.

It is possible to associate several ACL aliases with a file. When cfagent checks a files with ACLs, it reads the existing ACL and compares it to the new one. Files are only modified if they do not conform to the specification in the cfengine program. Let's look at a complete example:

```
files:

  $(HOME)/myfile acl=acl_alias1 action=fixall

acl:

  { acl_alias1

  method:append
  fstype:solaris
  user:len:rwx
  }
```

ACLs are viewed in Solaris with the command '`getfacl`'. Suppose that, before running this program, our test-file had permissions

```
user:*:rwx
user:mark:rwx           #effective:r-x
group:*:r-x              #effective:r-x
mask:r-x
other:r-x
default_user:rw-
default_group:r--
```

```
default_mask:-w-
default_other:rwx
```

After the cfagent run, the ACL would become:

```
user:*:rwx
user:mark:rwx           #effective:r-x
user:len:rwx            #effective:r-x
group:*:r-x              #effective:r-x
mask:r-x
other:r-x
default_user:rw-
default_group:r--
default_mask:-w-
default_other:rwx
```

Suppose we wanted to to remove 'w' bit for user '`jacobs`', or make sure that it was never there.

```
{ acl_alias1

method:append
fstype:solaris
user:jacobs:-w
}
```

Note that the method used here is append. That means that, whatever other access permissions we might have granted on this file, the user '`jacobs`' (a known cracker) will have no write permissions on the file. Had we used the method `overwrite` above, we would have eliminated all other access permissions for every user and added the above. If we really wanted to burn '`jacobs`', we could remove all rights to the file like this

```
user:jacobs:noaccess
```

The keyword `noaccess` removes all bits. Note that this is not necessarily the same as doing a `-rwx`, since some filesystems, like DFS, have more bits than this. Then, if we want to forgive and forget, the ACLs may be removed for `jacobs` with the syntax

```
user:jacobs:default
```

In Solaris, files inherit default ACLs from the directory they lie in; these are modified by the umask setting to generate their own default mask.

DFS ACLs look a little different. They are examined with the '`acl_edit`' command or with

```
dcecp -c acl show <filename>
```

In order to effect changes to the DFS, you have to perform a DCE login to obtain authentication cookies. The user '`cell_admin`' is a special user account for administrating a local DFS cell. Suppose we have a file with the following DCE ACL:

```
mask_obj:r-x---
user_obj:rwxcid
user:cell_admin:r--c-- #effective:r-----
group_obj:r-x--d       #effective:r-x---
other_obj:r-x---
```

Now we want to add '`wx`' permissions for user '`cell_admin`', and add new entries with '`rx`' permissons for group `acct-admin` and user '`root`'. This is done with the following ACL alias:

```
{ acl_alias2

method:append
fstype:dfs
user:/.../iu.hioslo.no/cell_admin:wx
group:/.../iu.hioslo.no/acct-admin:rx
user:/.../iu.hioslo.no/root:rx
user:*:-x
}
```

The local cell name '/.../iu.hioslo.no' is required here. Cfagent can not presently change ACLs in other cells remotely, but if your cfengine program covers all of the cell servers, then this is no limitation, since you can still centralize all your ACLs in one place. It is just that the execution and checking takes place at distributed locations. This is the beauty of cfengine. After running cfagent, with the above program snippet, the ACL then becomes:

```
mask_obj:r-x---
user_obj:rwcid
user:cell_admin:rwxc-- #effective:r-x---
user:root:r-x---       #effective:r-x---
group_obj:r-x--d       #effective:r-x---
group:acct-admin:r-x---
other_bj:r-x---
```

For the sake of simplicity we have only used standard Unix bits 'rwx' here, but more complicated examples may be found in DFS. For example,

```
user:mark:+rwx,-cid
```

which sets the read, write, execute flags, but removes the control, insert and delete flags. In the DFS, files inherit the inital object ACL of their parent directory, while new directories inherit the initial container object.

The objects referred to in DFS as `user_obj`, `group_obj` and so forth refer to the owner of a file. i.e. they are equivalent to the same commands acting on the user who owns the file concerned. To make the cfengine user-interface less cryptic and more in tune with the POSIX form, we have dropped the '_obj' suffices. A user field of '*' is a simple abbreviation for the owner of the file.

A problem with any system of lists is that one can generate a sequence which does one thing, and then undoes it and redoes something else, all in the same contradictory list. To avoid this kind of accidental interaction, cfengine insists that each user has only one ACE (access control entry), i.e. that all the permissions for a given user be in one entry.

5 Using cfengine as a front end for cron

One of cfengine's strengths is its use of classes to identify systems from a single file or set of files. Many administrators think that it would be nice if the cron daemon also worked in this way. One possible way of setting up cron from a global configuration would be to use the cfengine `editfiles` facility to edit each cron file separately. A much better way is to use cfengine's time classes to work like a user interface for cron. This allows you to have a single, central cfengine file which contains all the cron jobs on your system without losing any of the fine control which cron affords you. All of the usual advantages apply:

- It is easier to keep track of what cron jobs are running on the system when you have everything in one place.
- You can use all of your carefully crafted groups and user-defined classes to identify which host should run which programs.

The central idea behind this scheme is to set up a regular cron job on every system which executes cfagent at frequent intervals. Each time cfagent is started, it evaluates time classes and executes the shell commands defined in its configuration file. In this way we use cfagent as a wrapper for the cron scripts, so that we can use cfengine's classes to control jobs for mulitple hosts. Cfengine's time classes are at least as powerful as `cron`'s time specification possibilities, so this does not restrict you in any way, See Section 5.3 [Building flexible time classes], page 60. The only price is the overhead of parsing the cfengine configuration file.

To be more concrete, imagine installing the following '`crontab`' file onto every host on your network:

```
#
# Global Cron file
#
0,15,30,45 * * * * /usr/local/sbin/cfexecd -F
```

5.1 Structuring '`cfagent.conf`'

The structure of '`cfagent.conf`' needs to reflect your policy for running jobs on the system. You need to switch on relevant tasks and switch off unwanted tasks depending on the time of day. This can be done in three ways:

- By placing individual actions under classes which restrict the times at which they are executed,

```
action:

     Hr00.Min10_15||Hr12.Min45_55::

        Command
```

- By choosing a different `actionsequence` depending on the time of day.

```
control:
```

```
Hr00::   # Action-sequence for daily run at midnight

   actionsequence = ( sequence )

!Hr00::   # Action-sequence otherwise

    actionsequence = ( sequence )
```

- By importing modules based on time classes.

```
import:

      Hr00:: cf.dailyjobs

      any::  cf.hourlyjobs
```

The last of these is the most efficient of the three, since cfengine does not even have to spend time parsing the files for actions which you know you will not want.

5.2 Splaying host times

The trouble with starting every cfagent at the same time using a global cron file is that it might lead to contention or inefficiency. For instance, if a hundred cfagents all suddenly wanted to copy a file from a master source simultaneously this would lead to a big load on the server. We can prevent this from happening by introducing a time delay which is unique for each host and not longer than some given interval. Cfagent uses a hashing algorithm to generate a number between zero and a maximum value in minutes which you define, like this:

```
# Put this in update.conf, so that the updates are also splayed

control:

   SplayTime = ( 10 ) # minutes
```

If this number is non-zero, cfagent goes to sleep after parsing its configuration file and reading the clock. Every machine will go to sleep for a different length of time, which is no longer than the time you specify in minutes. A hashing algorithm, based on the fully qualified name of the host, is used to compute a unique time for hosts. The shorter the interval, the more clustered the hosts will be. The longer the interval, the lighter the load on your servers. This 'splaying' of the run times will lighten the load on servers, even if they come from domains not under your control but have a similar cron policy.

Splaying can be switched off temporarily with the '`-q`' or '`--no-splay`' options.

5.3 Building flexible time classes

Each time cfagent is run, it reads the system clock and defines the following classes based on the time and date:

`Yrxx::` The current year, e.g. '`Yr1997`', '`Yr2001`'. This class is probably not useful very often, but it might help you to turn on the new-year lights, or shine up your systems for the new millenium!

Month:: The current month can be used for defining very long term variations in the system configuration, e.g. '`January`', '`February`'. These classes could be used to determine when students have their summer vacation, for instance, in order to perform extra tidying, or to specially maintain some administrative policy for the duration of a conference.

Day:: The day of the week may be used as a class, e.g. '`Monday`', '`Sunday`'.

`Day`*xx*:: A day in the month (date) may be used to single out by date, e.g. the first day of each month defines '`Day1`', the 21st '`Day21`' etc.

`Hr`*xx*:: An hour of the day, in 24-hour clock notation: '`Hr00`'...'`Hr23`'.

`Min`*xx*:: The precise minute at which cfagent was started: '`Min0`' ... '`Min59`'. This is probably not useful alone, but these values may be combined to define arbitrary intervals of time.

`Min`*xx*`_`*xx*::
The five-minute interval in the hour at which cfagent was executed, in the form '`Min0_5`', '`Min5_10`' .. '`Min55_0`'.

Time classes based on the precise minute at which cfagent started are unlikely to be useful, since it is improbable that you will want to ask cron to run cfagent every single minute of every day: there would be no time for anything to complete before it was started again. Moreover, many things could conspire to delay the precise time at which cfagent were started. The real purpose in being able to detect the precise start time is to define composite classes which refer to arbitrary intervals of time. To do this, we use the `group` or `classes` action to create an alias for a group of time values. Here are some creative examples:

```
classes:  # synonym groups:

  LunchAndTeaBreaks = ( Hr12 Hr10 Hr15 )

  NightShift        = ( Hr22 Hr23 Hr00 Hr01 Hr02 Hr03 Hr04 Hr05 Hr06 )

  ConferenceDays    = ( Day26 Day27 Day29 Day30 )

  QuarterHours      = ( Min00 Min15 Min30 Min45 )

  TimeSlices        = ( Min01 Min02 Min03 Min33 Min34 Min35)
```

In these examples, the left hand sides of the assignments are effectively the ORed result of the right hand side. This if any classes in the parentheses are defined, the left hand side class will become defined. This provides a flexible and readable way of specifying intervals of time within a program, without having to use '`|`' and '`.`' operators everywhere.

5.4 Choosing a scheduling interval

How often should you call your global cron script? There are several things to think about:

- How much fine control do you need? Running cron jobs once each hour is usually enough for most tasks, but you might need to exercise finer control for a few special tasks.

- Are you going to run the entire cfengine configuration file or a special light-weight file?
- System latency. How long will it take to load, parse and run the cfengine script?

Cfengine has an intelligent locking and timeout policy which should be sufficient to handle hanging shell commands from previous crons so that no overlap can take place, See Section 6.2.3 [Spamming and security], page 65.

6 Cfengine and network services

This chapter describes how you can set up a cfengine network service to handle remote file distribution and remote execution of cfengine without having to open your hosts to possible attack using the `rsh` protocols.

6.1 Cfengine network services

By starting the daemon called `cfservd`, you can set up a line of communication between hosts, allowing them to exchange files across the network or execute cfengine remotely on another system. Cfengine network services are built around the following components:

`cfagent` The configuration engine, whose only contact with the network is via remote copy requests. This component does the hard work of configuring the system based on rules specified in the file '`cfagent.conf`'. It does not and cannot grant any access to a system from the network.

`cfservd` A daemon which acts as both a file server and a remote-cfagent executor. This daemon authenticates requests from the network and processes them according to rules specified in '`cfservd.conf`'. It works as a file server and as a mechanism for starting cfagent on a local host and piping its output back to the network connection.

`cfrun` This is a simple initiation program which can be used to run cfagent on a number of remote hosts. It cannot be used to tell cfagent what to do, it can only ask cfagent on the remote host to run the configuration file it already has. Anyone could be allowed to run this program, it does not require any special user privileges. A locking mechanism in cfengine prevents its abuse by spamming.

`cfwatch` This program (which is not a part of the distribution: it is left for others to implement) should provide a graphical user interface for watching over the configuration of hosts running cfagent and logging their output.

With these components you can emulate programs like `rdist` whose job it is to check and maintain copies of files on client machines. You may also decide who has permission to run cfagent and how often it may be run, without giving away any special user privileges.

6.2 How it works

6.2.1 Remote file distribution

This section describes how you can set up `cfservd` as a remote file server which can result in the distribution of files to client hosts in a more democratic way than with programs like rdist.

An important difference between cfengine and other systems has to do with the way files are distributed. Cfengine uses a 'pull' rather than a 'push' model for distributing network files. The `rdist` command, for instance, works by forcing an image of the files on one server machine onto all clients. Files get changed when the server wishes it and the clients have no choice but to live with the consequences. Cfengine cannot force its will onto other hosts in this way, it can only signal them and ask them to collect files if they want to. In other words, cfengine simulates a 'push' model by polling each client and running the local cfengine configuration script giving the host the chance to 'pull' any updated files from the remote server, but leaving it up to the client machine to decide whether or not it wants to update.

Also, in contrast to programs like `rdist` which distribute files over many hosts, cfengine does not require any general `root` access to a system using the '`.rhosts`' file or the '`/etc/hosts.equiv`' file. It is sufficient to run the daemon as root. You can not run it by adding it to the '`/etc/inetd.conf`' file on your system however. The restricted functionality of the daemon protects your system from attempts to execute general commands as the root user using `rsh`.

To remotely access files on a server, you add the keyword `server=`*`host`* to a copy command. Consider the following example which illustrates how you might distribute a password file from a masterhost to some clients.

```
copy:

  PasswdClients::

    /etc/passwd  dest=/etc/passwd owner=root group=0 server=server-host
```

Given that the `cfservd` daemon is running on *server-host*, cfagent will make contact with the daemon and attempt to obtain information about the file. During this process, cfengine verifies that the system clocks of the two hosts are reasonably synchronized. If they are not, it will not permit remote copying. If cfagent determines that a file needs to be updated from a remote server it begins copying the remote file to a new file on the same filesystem as the destination-file. This file has the suffix '`.cfnew`'. Only when the file has been successfully collected will cfagent make a copy of the old file, (see `repository` in the Reference manual), and rename the new file into place. This behaviour is designed to avoid race-conditions which can occur during network connections and indeed any operations which take some time. If files were simply copied directly to their new destinations it is conceivable that a network error could interrupt the transfer leaving a corrupted file in place.

Cfagent places a timeout of a few seconds on network connections to avoid hanging processes.

Normally the daemon sleeps, waiting for connections from the network. Such a connection may be initiated by a request for remote files from a running cfengine program on another host, or it might be initiated by the program `cfrun` which simply asks the host running the daemon to run the cfengine program locally.

6.2.2 Remote execution of cfagent

It is a good idea to execute cfagent by getting `cron` to run it regularly. This ensures that cfagent will be run even if you are unable to log onto a host to run it yourself. Sometimes however you will want to run cfagent immediately in order to implement a change in configuration as quickly as possible. It would then be inconvenient to have to log onto every host in order to do this manually. A better way would be to issue a simple command which contacted a remote host and ran cfagent, printing the output on your own screen:

```
myhost% cfrun remote-host -v

 output....
```

A simple user interface is provided to accomplish this. `cfrun` makes a connection to a remote cfservd-daemon and executes cfagent on that system with the privileges of the cfservd-daemon (usually `root`). This has a two advantages:

- You avoid having to log in on a remote host in order to reconfigure it.
- Users other than root can run cfagent to fix any problems with the system.

A potential disadvantage with such a system is that malicious users might be able to run cfagent on remote hosts. The fact that non-root users can execute cfagent is not a problem in itself, after all the most malicious thing they would be able to do would be to check the system configuration and repair any problems. No one can tell cfagent what to do using the cfrun program, it is only possible to run an existing configuration. But a more serious concern is that malicious users might try to run cfagent repeatedly (so-called 'spamming') so that a system became burdened with running cfagent constantly, See Section 6.2.3 [Spamming and security], page 65.

6.2.3 Spamming and security

The term 'spamming' refers to the senseless repetition of something in a malicious way intended to drive someone crazy[1]. In the computer world some malicious users, a bit like 'flashers' in the park[2] like to run around the net a reveal themselves ad nauseum by sending multiple mail messages or making network connections repeatedly to try to overload systems and people[3].

Whenever we open a system to the network, this problem becomes a concern. Cfengine is a tool for making peace with networked systems, not a tool to be manipulated into acts of senseless aggression. The cfservd daemon does make it possible for anyone to connect and run a cfengine process however, so clearly some protection is required from such attacks.

Cfengine's solution to this problem is a locking mechanism. Rather than providing user-based control, cfengine uses a time based locking mechanism which prevents actions from being executed unless a certain minimum time has elapsed since the last time they were executed. By using a lock which is not based on user identity, we protect several interests in one go:

- Restricting cfengine access to root would prevent regular users, in trouble, from being able to fix problems when the system administrator was unavailable. A time-based lock does not prevent this kind of freedom.
- Accidents with cron or shell scripts could start cfagent more often than desirable. We also need to protect against such accidents.
- We can prevent malicious attacks regardless of whom they may come from.

Cfengine is controlled by a series of locks which prevent it from being run too often, and which prevent it from spending too long trying to do its job. The locks work in such a way

---

[1] Recall the 'spam' song from Monty Python's flying circus?

[2] Recall the 'spam' song from Monty Python's flying circus?

[3] Recall the 'spam' song ... get the idea?

that you can start several cfengine processes simultaneously without them crashing into each other. Coexisting cfengine processes are also prevented from trying to do the same thing at the same time (we call this 'spamming'). You can control two things about each kind of action in the action sequence:

- The minimum time which should have passed since the last time that action was executed. It will not be executed again until this amount of time has elapsed. (Default time is 1 minute.)
- The maximum amount of time cfagent should wait for an old instantiation of cfagent to finish before killing it and starting again. (Default time is 120 minutes.)

You can set these values either globally (for all actions) or for each action separately. If you set global and local values, the local values override the global ones. All times are written in units of *minutes*.

```
 actionsequence
   (
   action.IfElapsedtime-in-mins
   action.ExpireAftertime-in-mins
   )
```

or globally,

```
control:

   IfElapsed   = ( time-in-mins )

   ExpireAfter = ( time-in-mins )
```

For example:

```
control:

  actionsequence =
    (
    files.IfElapsed240.ExpireAfter180
    copy
    tidy
    )

  IfElapsed = ( 30 )
```

In this example, we treat the files action differently to the others. For all the other actions, cfagent will only execute the files part of the program if 30 minutes have elapsed since it was last run. Since no value is set, the expiry time for actions is 60 minutes, which means that any cfagent process which is still trying to finish up after 60 minutes will be killed automatically by the next cfagent which gets started.

As for the files action: this will only be run if 240 minutes (4 hours) have elapsed since the last run. Similarly, it will not be killed while processing 'files' until after 180 minutes (3 hours) have passed.

These locks do not prevent the whole of cfagent from running, only so-called 'atoms'. Several different atoms can be run concurrently by different cfagents. Assuming that the time conditions set above allow you to start cfengine, the locks ensure that atoms will never be started by two cfagents at the same time, causing contention and wasting CPU cycles. Atoms are defined to maximize the security of your system and to be efficient. If cfengine were to lock each file it looked at seperately, it would use a large amount of time processing the locks, so it doesn't do that. Instead, it groups things together like this:

`copy, editfiles, shellcommands`
: Each separate command has its own lock. This means that several such actions can be processed concurrently by several cfagent processes. Multiple or recursive copies and edits are treated as a single object.

`netconfig, resolve, umount, mailcheck, addmounts, disable, processes`
: All commands of this action-type are locked simultaneously, since they can lead to contention.

`mountall, mountinfo, required, checktimezone`
: These are not locked at all.

Cfagent creates a directory '`~/.cfengine`' for writing lock files for ordinary users.

The option '`-K`' or '`--no-lock`' can be used to switch off the locking checks, but note that when running cfagent remotely via `cfservd`, this is not possible.

6.2.4 Some points on the cfservd protocol

Cfservd uses a form for host-based authorization. Each atomic operation, such as statting, getting files, reading directories etc, requires a new connection and each connection is verified by a double reverse lookup in the server's DNS records. Single stat structures are cached during the processing of a file.

MD5 checksums are transferred from client to server to avoid loading the server. Even if a user could corrupt the MD5 checksum, he or she would have to get past IP address access control and the worst that could happen would be to get the right version of the file. Again this is in keeping with the idea that users can only harm themselves and not others with cfengine.

6.2.5 Deadlocks and runaway loops

Whenever we allow concurrent processes to share a resource, we open ourselves up the possibilty of deadlock. This is a situation where two or more processes are locked in a vicious stalemate from which none can escape. Another problem is that it might be possible to start an infinite loop: cfagent starts itself.

Cfagent protects you from such loops to a large degree. It should not be possible to make such a loop by accident. The reason for this is the locking mechanism which prevents tasks being repeated too often. If you start a cfagent process which contains a shell-command to start cfagent again, this shell command will be locked, so it will not be possible to run it a second time. So while you might be able to start a second cfagent process, further processes will not be started and you will simply have wasted a little CPU time. When the first cfagent returns, the tasks which the second cfagent completed will not be repeated unless you have set the `IfElapsed` time or the `ExpireAfter` time to zero. In general, if you wish to avoid problems like this, you should not disable the locking mechanism by setting these two times to zero.

The possibility of deadlock arises in network connection. Cfengine will not attempt to use the network to copy a file which can be copied internally from some machine to itself. It will always replace the `server=` directive in a copy with 'localhost' to avoid unnecessary network connections. This prevents one kind of deadlock which could occur: namely cfrun executes cfagent on host A (cfservd on host A is then blocked until this completes), but the host A configuration file contains a remote copy from itself to itself. This remote copy would then have to wait for cfservd to unblock, but this would be impossible since cfservd cannot unblock until it has the file. By avoiding remote copies to localhost, this possibility is avoided.

6.3 Configuring `cfservd`

6.3.1 Installation of `cfservd`

To install the cfservd daemon component, you will need to register a port for cfengine by adding the following line to the system file '`/etc/services file`'

```
cfengine        5308/tcp
```

You could do this for all hosts by adding the following to your cfengine configuration

```
editfiles:

  { /etc/services

   AppendIfNoSuchLine "cfengine        5308/tcp"
  }
```

To start cfservd at boot time, you need to place a line of the following type in your system startup files:

```
# Start cfengine server
cfservd
```

Note that `cfservd` will re-read its configuration file whenever it detects that it has been changed, so you should not have to restart the daemon, not send it the HUP signal as with other daemons.

6.3.2 Configuration file '`cfservd.conf`'

The server daemon is controlled by a file called '`cfservd.conf`'. The syntax of this configuration file is deliberately modelled on cfengine's own configuration file, but despite the similarities, you cannot mix the contents of the two files.

Though they are not compatible, '`cfagent.conf`' and '`cfservd.conf`' are similar in several ways:

- Both files use classes to label entries, so that you may use the same configuration file to control all hosts on your network. This is a convenience which saves you the trouble of maintaining many different files.

- Both files are searched for using the contents of the variable `CFINPUTS`.

- You can use `groups` and `import` in both files to break up files into convenient modules and to import common resources, such as lists of groups.

Note that the classes in the '`cfservd.conf`' file do not tell you the classes of host which have access to files and directories, but rather which classes of host pay attention to the access and deny commands when the file is parsed.

Host name authentication is not by class or group but by hostname, like the '`/etc/exports`' file on most Unix systems. The syntax for the file is as follows:

```
control:

  classes::

      domain = ( DNS-domain-name )

      cfrunCommand = ( "script/filename" )  # Quoted

      MaxConnections = ( maximum number of forked daemons )

      ChecksumDatabase = ( filename )

      IfElapsed = ( time-in-minutes )

      DenyBadClocks = ( false )

      AllowConnectionsFrom = ( IP numbers )

      DenyConnectionsFrom = ( IP numbers )

      AllMultipleConnectionsFrom = ( IP numbers )

      LogAllConnections = ( false/true )

      SkipVerify = ( IP numbers )

groups:

  Group definitions

import:

  Files to import

admit: | grant:

  classes::

     /file-or-directory

       wildcards/hostnames

deny:

  classes::

     /file-or-directory

       wildcards/hostnames root=hostlist encrypt=true/on
```

See the reference manual for descriptions of these elements. The file consists of a control section and access information. You may use the control section to define any variables which you want to use, for convenience, in the remainder of your file.

Following the control section, comes a list of files or directories and hosts which may access these. If permissions are granted to a directory then all subdirectories are automatically granted also. Note that plain-file symbolic links are not checked for, so you may need to specifically deny access to links if they are plain files, but cfservd does not follow directory symbolic links and give access to files in directories pointed to by these.

Fully qualified hostnames should be used in this file. If host names are unqualified, the current domain is appended to them (do not forget to define the domain name). Authentication calls the Unix function `gethostbyname()` and so on to identify and verify connecting hosts, so the names in the file must reflect the type on names returned by this function. You may use wildcards in names to match, for instance, all hosts from a particular domain.

Here is an example file

```
#####################################################
#
# This is a cfservd config file
#
#####################################################

groups:

  PasswdHost = ( nexus )

#####################################################

control:

  #
  # Assuming CFINPUTS is defined
  #

  cfrunCommand = ( "/var/cfengine/bin/cfagent" )

  variable = ( /usr/local/publicfiles )

#####################################################

admit:   # Can also call this grant:


    # Note that we must grant access to the
    # agent if we wan to start it remotely with cfrun

      /var/cfengine/bin/cfagent

   PasswdHost::

     /etc/passwd

        *.iu.hioslo.no

    FtpHost::

    # An alternative to ftp, grant anyone

       /local/ftp/pub

         *

    # These file paths must not contain symbolic
    # links. Access control does not follow symlinks.
```

```
    any::

       $CFINPUTS/cfrun.sh

         *.iu.hioslo.no

#####################################################

deny:

   /etc/services

       borg.iu.hioslo.no

  /local/ftp

       *.pain-in-the-ass.com
```

NOTE I: cfservd is not `rpc.mountd`*, access control is by filename, not by device name. Do not assume that files lying in subdirectories are not open for access simply because they lie on a different device. You should give the real path name to file and avoid symbolic links.*

NOTE II: access control is per host and per user. User names are assumed to be common to both hosts. There is an implicit trust relationship here. There is no way to verify whether the user on the remote host is the same user as the user with the same name on the local host.

NOTE III: Cfservd requires you to grant access to files without following any symbolic links. You must grant access to the real file or directory in order to access the file object. This is a security feature in case parties with login access to the server could grant access to additional files by having the permission to create symbolic links in a transitory directory, e.g. '`/tmp`'.

7 Security and cfengine

Computer security is about protecting the data and availability of an association of hosts. Briefly, the key words are authentication, privacy, integrity and trust. To understand computer security we have to understand the interrelationships between all of the hosts and services on our networks as well as the ways in which those hosts can be accessed. Tools which allow this kind of management are complex and usually expensive.

7.1 What is security?

For a computer to be secure it must be *physically secure* — if we can get our hands on a host then we are never more than a screwdriver away from all of its assets—but assuming that hosts are physically secure, we then wish to deal with the issues of software security which is a much more difficult topic. Software security is about access control and software reliability. No single tool can make computer systems secure. Major blunders have been made out of the belief that a single product (e.g. a 'firewall') would solve the security problem. For instance, a few years ago a cracker deleted all the user directories from a dialup login server belonging to a major Norwegian telecommunications company, from the comfort of his web browser. This was possible, even through a firewall, because the web server on the host concerned was incorrectly configured. The bottom line is that there is no such thing as a secure operating system, firewall or none. What is required is a persistent mixture of vigilence and adaptability.

For many, security is perceived as being synonymous with network privacy or network intrusion. Privacy is one aspect of security, but it is not the network which is our special enemy. Many breaches of security happen from within. There is little difference between the dangers of remote access from the network or direct access from a console: privacy is about access control, no matter where the potential intruder might be. If we focus exclusively on network connectivity we ignore a possible threat from internal employees (e.g. the janitor who is a computer expert and has an axe to grind, or the mischievous son of the director who was left waiting to play mom's office, or perhaps the unthinkable: a disgruntled employee who feels as though his/her talents go unappreciated). Software security is a vast subject, because modern computer systems are complex. It is only exacerbated by the connectivity of the internet which allows millions of people to have a go at breaking into networked systems. What this points to is the fact that a secure environment requires a tight control of access control on every host individually, not merely at specific points such as firewalls.

This article is not a comprehensive guide to security. Rather it is an attempt to illustrate how cfengine can be used to help you automate a level of host integrity on all the hosts of your network. Cfengine is a network configuration tool with two facets. It is a language used to build an 'expert system'. An expert system describes the way you would like your hosts and network to look and behave. CFengine is also a software robot which compares the model you have described with what the world really looks like and then sets to work correcting any deviations from that picture. In many ways it is like an immune system, neutralizing and repairing damaged parts. Unlike many shell-script packages for sysadmin, cfengine is a C program which means light on system resources. Also it works on a principle of 'convergence'. Convergence means that each time you run cfengine the system should get closer to the model which you have described until eventually when the system is the model, cfengine becomes quiescent, just like an immune system. In the words of one user your hosts 'never get worse'. This assumes of course that the model you have is what you really want. Using cfengine, model building becomes synonymous with formulating and formalizing a system policy.

What makes cfengine a security tool is that security policy is a part of system policy: you cannot have one without the other. You will never have security unless you are in control of your network. Cfengine monitors and indeed repairs hosts with simple easily controllable actions. From an automation perspective, security is no different from the general day to day business of system maintenance, you just need to pay more attention to the details. We cannot speak of 'have security' and 'have not security'. There is always security, it is simply a matter of degree: weak or strong; effective or ineffective.

7.2 A word of warning

Before starting it is only proper to state the obvious. You should never trust anyone's advice about configuration or security without running it past your own grey matter first. The examples provided here are just that: examples. They might apply to you as written and they might need to be modified. You should never accept and use an example without thinking carefully and critically first! Also, in any book of recipies or guide to successful living you know that there are simplified answers to complex questions and you should treat them as such. There is no substitute for real understanding.

7.3 Automation

Even in the smallest local area network you will want to build a scheme for automating host configuration and maintenance, because networks have a way of growing from one host into many quite quickly. It is therefore important to build a model which scales. A major reason for using cfengine is precisely for scalability. Whether you have one host or a hundred makes little difference. Cfengine is instructed from a central location, but its operation is completely and evenly spread across the network. Each host is responsible for obtaining a copy of the network model from a trusted source and is then responsible for configuring itself without intervention from outside. Unlike some models, cfengine does not have to rely on network communication or remote object models.

We also need integration, or the ability to manage the interrelationships between hosts. It is no good having complete control of one important host and thinking that you are secure. If an intruder can get into any host, he or she is almost certain to get into the ones that matter, especially if you are not looking at all of them. Using cfengine is a good way of forcing yourself to formulate a configuration/security policy and then stick to it. Why cfengine? There are three reasons: i) it forces a discipline of preparation which focuses you on the problems at the right level of detail, ii) it provides you with 'secure' scalable automation and a common interface to all your hosts, and iii) it scales to any number of hosts without additional burdens. We'll need to qualify some of these points below.

The first step in security management is to figure out a security policy. That way, you know what *you* mean by security and if that security is breached, you will know what to do. In many cases you can formulate a large part of your security policy as cfengine code. That makes it formal, accurate and it means that it will get done by the robot without requiring any more work on your part.

As an immune system, cfengine will even work fine in a partially connected environment it makes each host responsible for its own state. It is not reliant on network connectivty for remote method invocations or CORBA-style object requests as is, say, Tivoli. All it needs is an authentic copy of the network configuration document stored locally on each host. If this is the case, a detached host will not be left unprotected, at worst it might lag behind in its version of the network configuration.

7.4 Trust

There are many implicit trust relationships in computer systems. It is crucial to understand them. If you do not understand where you are placing your trust, your trust can be exploited by attackers who have thought more carefully than you have.

For example, any NFS server of users' home-directories trusts the root user on the hosts which mount those directories. Some bad accidents are prevented by mapping root to the user nobody on remote systems, but this is not security, only convenience. The root user can always use 'su' to becomes any user in its password file and access/change any data within those filesystems. The .rlogin and hosts.equiv files on Unix machines grant root (or other user) privileges to other hosts without the need for authentication.

If you are collecting software from remote servers, you should make sure that they come from a machine that you trust, particularly if they are files which could lead to privileged access to your system. Even checksums are no good unless they also are trustworthy. For example, it would be an extremely foolish idea to copy a binary program such as /bin/ps from a host you know nothing about. This program runs with root privileges. If someone were to replace that version of ps with a Trojan horse command, you would have effectively opened your system to attack. Most users trust anonymous FTP servers where they collect free software. In any remote copy you are setting up an implicit trust relationship. First of all you trust integrity of the host you are collecting files from. Secondly you trust that they have the same username database with regard to access control. The root user on the collecting host has the same rights to read files as the root user on the server. The same applies to any matched user name.

7.5 Why trust cfengine?

Cfengine has a very simple trust model. It trusts the integrity of its input file and any data which is explicitly chooses to download. Cfengine places the responsibility on root on the localhost not on any outsiders. *You* can make cfengine destroy your system, just as you can destroy it yourself, but no one else can, so as long as you are careful with the input file you are trusting essentially no-one. We shall qualify this below for remote file copying.

Cfengine assumes that its input file is secure. Apart from that input file, no part of cfengine accepts or uses any configuration information from outside sources. The most one could do from an authenticated network connection is to ask cfengine to carry out (or not) certain parts of its model, thus in the worst case scanario an outside attacker could spoof cfengine into configuring the host correctly. In short, no one except root on the localhost can force cfengine to do anything (unless root access to your system has already been compromised by another route). This means that there is a single point of failure. The input file does not even have to be private as long as it is authentic. No one except you can tell cfengine what to do.

There is a catch though. Cfengine can be used to perform remote file transfer. In remote file transfer one is also forced to trust the integrity of the data received, just as in any remote copy scheme. Although cfengine works hard to authenticate the identity of the host, once the host's identity is verified it cannot verify the accuracy of unknown data it has been asked to receive. Also, as with all remote file transfers, cfengine could be tricked by a DNS spoofing into connecting to an imposter host, so use the IP addresses of hosts, not their names if you don't trust your DNS service. In short, these faults are implicit in remote copying. They do not have to do with cfengine itself. This has nothing to do with encryption as users sometimes believe: encrypted connections do not change these trust relationships—they improve the privacy of the data being transmitted not their accuracy or trustworthiness.

The point of cfengine is normally to have only one global configuration for every host. This needs to be distributed somehow which means that hosts must collect this file from a remote server. This in turn means that you must trust the host which has the master copy of the cfengine configuration file.

7.6 Configuration

The beginning of security is correct host configuration. Even if you have a firewall shielding you from outside intrusion, an incorrectly configured host is a security risk. Host configuration is what cfengine is about, so we could easily write a book on this. Rather than reiterating the extensive documentation, let's just consider a few examples which address actual problems and get down to business without further ado.

A cfengine configuration file is composed of objects with the following syntax (see the cfengine documentation):

```
rule-type:

  classes-of-host-this-applies-to::

        Actual rule 1
        Actual rule 2 ...
```

The rule-types include checking file permissions, editing textfiles, disabling (renaming and removing permissions to) files, controlled execution of scripts and a variety of other things relating to host configuration. Some of the 'control' rules are simply flags which switch on complex (read 'smart') behaviour. Every cfengine program needs an actionsequence which tells it the order in which bulk configuration operations should be evaluated. e.g.

```
control:

  actionsequence = ( netconfig copy processes editfiles )
```

You should look at the cfengine manual to get started with your configuration.

Let us step through some basic idioms which can repeated in different contexts.

As representative examples we shall take Solaris and GNU/Linux as example operating systems. This is not to single them out as being particularly secure or insecure, it is merely due to their widespread use and for definiteness.

7.7 Disabling and replacing software

One of the simplest things which we are asked to do constantly is to disable dangerous programs as bugs are discovered. CERT security warnings frequently warn about programs with flaws which can compromise a system. In cfengine, disabling a file means renaming it to *.cf-disabled and setting its permission to 400.

```
disable:

  #
  # CERT security patches
  #

  solaris::

    /usr/openwin/bin/kcms_calibrate
    /usr/openwin/bin/kcms_configure
    /usr/bin/admintool
    /etc/rc2.d/S99dtlogin
    /usr/lib/expreserve

  linux::

     /sbin/dip-3.3.7n
     /etc/sudoers
     /usr/bin/sudoers
```

Although this is a trivial matter, the fact that it is automated means that cfengine is checking for this all the time. As long as a host is up and running (connected to the network or not) cfengine will be ensuring the named file is not present.

Another issue is to replace standard vendor programs with drop-in replacements. For example, most admins would like to replace their vendor sendmail with the latest update from Eric Allman's site. One way to do this is to compile the new sendmail into a special directory, separate from vendor files and then to symbolically link the new program into place.

```
links:

  solaris||linux::

   /usr/lib/sendmail      ->!  /usr/local/lib/mail/bin/sendmail-8.9.3
   /usr/sbin/sendmail     ->!  /usr/local/lib/mail/bin/sendmail-8.9.3
   /etc/mail/sendmail.cf  ->!  /usr/local/lib/mail/etc/sendmail.cf
```

The exclamation marks mean (by analogy with the csh) that existing file objects should be replaced by links to the named files. Again the integrity of these links is tested every time cfengine runs. If the object /usr/lib/sendmail is not a link to the named file, the old file is moved and a link is made. If the link is okay, nothing happens. After putting the new sendmail in place, you will need to make sure that the restricted shell configuration is in order.

```
  #
  # Sendmail, restricted shell needs these links
  #

  solaris::
```

```
  # Most of these will only be run on the MailHost
  # but flist (procmail) is run during sending...

  /usr/adm/sm.bin/vacation -> /usr/ucb/vacation
  /usr/adm/sm.bin/flist   ->  /home/listmgr/.bin/flist

linux::

  /usr/adm/sm.bin/vacation -> /usr/bin/vacation
```

Link management is a particularly useful feature of cfengine. By putting links (actually all system modifications) into the cfengine configuration and never doing anything by hand, you build up a system which is robust to reinstallation. If you lose your host, you just have to run cfengine once or twice to reconstruct it.

Of course, the fundamental tenet of security is to be able to restrict privilege to resources. We therefore need to check the permissions on files. For instance, a recent CERT advisory warned of problems with some free unix mount commands which were setuid root. If we suppose there is a group of hosts called 'securehosts' which we don't need to worry about, then we could remove the setuid bits on all other hosts as follows:

```
files:

  !securehosts.linux::

     /bin/mount     mode=555 owner=root action=fixall
     /bin/umount    mode=555 owner=root action=fixall

  securehosts.linux::

     /bin/mount      m=6555 o=root action=fixall
     /bin/umount     m=6555 o=root action=fixall
```

One area where cfengine excels over other tools is in its ascii file editing abilities. Editing textfiles in a non-destructive way is such an important operation that having used it you will wonder how you every managed without it! Here are some simple but real examples of how file editing can be used.

```
editfiles:

  # sun4, who are they kidding?

  { /etc/hosts.equiv

  HashCommentLinesContaining "+"
  }

  #
  # CERT security patch for vold vulnerability
  #

  sunos_5_4::

     { /etc/rmmount.conf

     HashCommentLinesContaining "action cdrom"
     HashCommentLinesContaining "action floppy"
```

```
     }
```

TCP wrapper configuration can be managed easily by maintaining a pair of master files on a trusted host. Files of the form

```
# /etc/hosts.allow (exceptions)
#
# Public services

sendmail: ALL
in.ftpd:  ALL
sshd:     ALL

# Private services

in.fingerd:  .example.org LOCAL
in.cfingerd: .example.org LOCAL
sshdfwd-X11: .example.org LOCAL

# Portmapper has to use IP series

portmap: 128.39.89. 128.39.74. 128.39.75.
```

and

```
# /etc/hosts.deny (default)

ALL: ALL
```

may be distributed to each host by cfengine

```
copy:

 /masterfiles/hosts.deny dest=/etc/hosts.deny
                         mode=644
                         server=trusted
 /masterfiles/hosts.allow dest=/etc/hosts.allow
                          mode=644
                          server=trusted
```

and installed as follows

```
editfiles:

     { /etc/inet/inetd.conf

     # Make sure we're using tcp wrappers

     ReplaceAll "/usr/sbin/in.ftpd"    With "/local/sbin/tcpd"
     ReplaceAll "/usr/sbin/in.telnetd" With "/local/sbin/tcpd"
     ReplaceAll "/usr/sbin/in.rshd"    With "/local/sbin/tcpd"
     ReplaceAll "/usr/sbin/in.rlogind" With "/local/sbin/tcpd"

processes:

     "inetd" signal=hup
```

The services which we do not need should be removed altogether. There's no sense in tempting fate:

```
editfiles:

      { /etc/inetd.conf

      # Eliminate unwanted services

      HashCommentLinesContaining "rwall"
      HashCommentLinesContaining "/usr/sbin/in.fingerd"
      HashCommentLinesContaining "comsat"
      HashCommentLinesContaining "exec"
      HashCommentLinesContaining "talk"
      HashCommentLinesContaining "echo"
      HashCommentLinesContaining "discard"
      HashCommentLinesContaining "charge"
      HashCommentLinesContaining "quotas"
      HashCommentLinesContaining "users"
      HashCommentLinesContaining "spray"
      HashCommentLinesContaining "sadmin"
      HashCommentLinesContaining "rstat"
      HashCommentLinesContaining "kcms"
      HashCommentLinesContaining "comsat"
      HashCommentLinesContaining "xaudio"
      HashCommentLinesContaining "uucp"
      }
```

7.8 Process monitoring

When it comes to process management we are usually interested in three things: i) making sure certain processes are running, ii) making sure some processes are NOT running and iii) sending HUP signals to force configuration updates. To HUP a daemon and make sure that it is running, we write

```
processes:

 linux::

  "inetd"  signal=hup restart "/usr/sbin/inetd"   useshell=false
  "xntp"              restart "/local/sbin/xntpd" useshell=false
```

The useshell option tells cfengine that it should not use a shell to start the program. The idea here is to protect against IFS attacks. Unfortunately some programs require a shell in order to be started, but most do not. This is an extra precaution. When the cron daemon crashes, restarting it can be a problem since it does not close its filed descriptors properly when forking. The dumb-option helps here:

```
  "cron" matches=>1 restart "/etc/init.d/cron start"  useshell=dumb
```

To kill processes which should not be running, we write:

```
 processes:

   solaris::

   #
   # Don't want CDE stuff or SNMP peepholes...
```

```
   #

    "ttdbserverd" signal=kill
    "snmpd"       signal=kill
    "mibiisa"     signal=kill
```

A couple of years ago, a broken cracked account was revealed at Oslo College by the following test in the cfengine configuration:

```
processes:

   # Ping attack ?

   "ping"  signal=kill inform=true
```

There are few legimate reasons to run the ping command more than a few times. The chances of cfengine detecting single pings is quite small. But coordinated ping attacks are another story. When it was revealed that a user had twenty ping processes attempting to send large ping packets to hosts in the United States it was obvious the the account had been compromised. Fortunately for the recipient, the ping command was incorrectly phrased and would probably not have been noticed.

```
processes:

    "sshd"
                restart "/local/sbin/sshd"
                useshell=false

    "snmp"      signal=kill
    "mibiisa"   signal=kill

    "named"     matches=>1
                restart "/local/bind/bin/named"
                useshell=false

    # Do the network community a service and run this

    "identd"   restart "/local/sbin/identd" inform=true
```

Process management also includes the garbage collection which we shall return to briefly.

7.9 Monitoring files

Almost all security programs available are for the monitoring of file integrity. Cfengine also incorporates tools for monitoring files. Here are some of the elements in the faily complex files command:

```
files:

    classes::

       /file-object
                        mode=mode
                        owner=uid-list
                        group=gid-list
                        action=fixall/warnall..
```

```
                              ignore=pattern
                              include=pattern
                              exclude=pattern
                              checksum=md5
                              syslog=true/on/false/off
```

In additions to these, there are extra flags for BSD filesystems and ways of managing file ACLs for systems like NT. Here are some examples of basic checks on file permissions:

```
classes:

 # Define a class of hosts based on a test...

 have_shadow = ( '/bin/test -f /etc/shadow' )

 NFSservers = ( server1 server2 )

files:

  any::

     /etc/passwd mode=0644 o=root  g=other  action=fixplain

  have_shadow::

     /etc/shadow mode=0400 o=root  g=other  action=fixplain

  # Takes a while so do this at midnight and only on servers

  NFSservers.Hr00::

     /usr/local
           mode=-0002   Check no files are writable!
           recurse=inf
           owner=root,bin
           group=0,1,2,3,4,5,6,7,staff
           action=fixall
```

In the last example we parse through a whole file system (recurse=inf) and as a result we get a number of checks for free. Any previously unknown setuid programs are reported as well as any suspicious filenames (see below).

7.10 The setuid log

Cfengine is always on the lookout for files which are setuid or setgid root. It doesn't go actively looking for them uninvited, but whenever you get cfagent to check a file or directory with the files feature, it will make a note of setuid programs it finds there. These are recorded in the file cfengine.host.log which is stored under /var/cfengine or /var/log/cfengine. When new setuid programs are discovered, a warning is printed, but only if you are root. If you ever want a complete list, delete the log file and cfengine will think that all of the setuid programs it finds are new. The log file is not readable by normal users.

7.11 Suspicious filenames

Whenever cfagent opens a directory and scans through files and directories (recursively) (files, tidy, copy), it is also on the lookout for for suspicious filenames, i.e. files like ".. ." containing only space and/or dots. Such files are seldom created by sensible sources, but are often used by crackers to try to hide dangerous programs. Cfagent warns about such files. Although not necessarily a security issue, cfagent will also warn about filenames which contain non-printable characters if desired, and directories which are made to look like plain files by giving them filename extensions.

```
control:

   #
   # Security checks
   #

   NonAlphaNumFiles = ( on )
   FileExtensions = ( o a c gif jpg html ) # etc
   SuspiciousNames = ( .mo lrk3 lkr3 )
```

The file extension list may be used to detect concealed directories during these searches, if users create directories which look like common files this will be warned about. Additional suspicious filenames can be checked for automatically as a matter if course. This is commented further below.

The mail spool directory is a common place for users to try to hide dowloaded files. These options inform about files which do not have the name of a user or are not owned by a valid user:

```
 control:

   WarnNonOwnerMail = ( true )
   WarnNonUserMail = ( true )  # Warn about mail which is not owned by a user
```

Corresponding commands exist to delete these files without further ado. This can be a useful way of cleaning up after users whose accounts have been removed.

7.12 Checksums and Tripwire functionality

Cfagent can be used to check for changes in files which only something as exacting as an MD5 checksum/digest can detect. If you specify a checksum database and activate checksum verification,

```
control:

  ChecksumDatabase = ( /var/cfengine/cache.db )

  ChecksumUpdates = ( false )

files:

    /filename checksum=md5 ....
    /dirname  checksum=md5 recurse=inf....

    # If the database isn't secure, nothing is secure...
```

```
/var/cfengine/cache.db  mode=600 owner=root action=fixall
```

then cfagent will build a database of file checksums and warn you when files' checksums change. This makes cfagent act like Tripwire (currently only with MD5 checksums). It can be used to show up Trojan horse versions of programs. It should be used sparingly though since database management and MD5 checksum computation are resource intensive operations and this could add significant time to a cfagent run. The ChecksumUpdates variable (normally false) can be set to true to update the checksum database when programs change for valid reasons.

Warnings are all every fine and well, but the spirit of cfengine is not to bother us with warnings, it is to fix things automatically. Warning is a useful supplement, but in security breaches it is better to fix the problem, rather than leaving the host in a dangerous state. If you are worried about the integrity of the system then don't just warn about checksum mismatches here, make an md5 copy comparison against a read-only medium which has correct, trusted version of the file on it. That way if a binary is compromised you will not only warn about it but also repair the damage immediately!

The control variable ChecksumUpdates may be switched to on in order to force cfagent to update its checksum database after warning of a change.

7.13 FileExtensions

This list may be used to define a number of extensions which are regarded as being plain files by the system. As part of the general security checking cfagent will warn about any directories which have names using these extensions. They may be used to conceal directories.

```
FileExtensions = ( c o gif jpg html )
```

7.14 NonAlphaNumFiles

If enabled, this option causes cfagent to detect and disable files which have purely non-alphanumeric filenames, i.e. files which might be accidental or deliberately concealed. The files are then marked with a suffix .cf-nonalpha and are rendered visible.

```
NonAlphaNumFiles = ( on )
```

These files can then be tidied (deleted) or disabled by searching for the suffix pattern. Note that alphanumeric means ascii codes less than 32 and greater than 126.

7.15 Defensive garbage collection

We tend to be worried about the fact that crackers will destroy our systems and make them unusable, but many operating systems are programmed to do this to themselves! There are few systems which can survive a full system disk and yet many logging agents go on filling up disks without ever checking to see how full they are getting. In short they choke themselves in a self-styled denial of service attack. Cfagent can help here by rotating logs frequently and by tidying temporary file directories:

```
disable:
```

```
  Tuesday.Hr00::

   #
   # Disabling these log files weekly prevents them from
   # growing so enormous that they fill the disk!
   #

   /local/iu/httpd/logs/access_log   rotate=2
   /local/iu/httpd/logs/agent_log    rotate=2
   /local/iu/httpd/logs/error_log    rotate=2
   /local/iu/httpd/logs/referer_log  rotate=2

  FTPserver.Sunday::

   /local/iu/logs/xferlog rotate=3

tidy:

    /tmp pattern=* age=1
```

Process garbage collection is just as important. There are lot's of reasons why process tables fill up with unterminated processes. One example is faulty X terminal software which does not kill its children at logout. Another is that programs like netscape and pine tend to go into loops from which they never return, gradually loading the system with an ever increasing glacial burden. Just killing old processes can cause your system to spring back from its ice age blues (hopefully without littering the system with too many dead mammoths or bronze age axe-bearers). If the host concerned has important duties then this lack of responsiveness can compromise key services. It also gives local users a way of carrying out denial of service attacks on the system.

If users always log out at the end of the day and log in again the day after then this is easy to address with cfengine. Here is some code to kill commonly hanging processes. Note that on BSD like systems process options "aux" are required to see the relevant processes:

```
processes:

  linux|freebsd|sun4::

      SetOptionString "aux"

  any::

  "Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec"

      signal=kill

      include=ftpd
      include=tcsh
      include=xterm
      include=netscape
      include=ftp
      include=pine
      include=perl
      include=irc
      include=java
      include=/bin/ls
      include=emacs
```

```
include=passwd
```

This pattern works like this: as processes become more than a day old they name of the month appears in the date of the process start time. These are matched by the regular expression. The include lines then filter the list of the processes further picking out lines which include the specified strings. On some BSD-like systems the default ps option string is "-ax" and you might need to reset it to something which adds the start date in order to make this work.

Another job for process management is to clean up processes which have hung, gone amok or which are left over from old logins. Here is a regular expression which detects non-root processes which have clocked up more than 100 hours of CPU time. This is a depressingly common phenomenon when a program goes into an infinite loop. It can starve other processes of resources in a very efficient denial of service attack.

```
any::

 #
 # Kill processes which have run on for too long e.g. 999:99 cpu time
 # Careful a pattern to match 99:99 will kill everything!
 #

 "[0-9][0-9][0-9][0-9]:[0-9][0-9]" signal=term exclude=root
      "[0-9][0-9][0-9]:[0-9][0-9]" signal=term exclude=root
```

Under NT this is not so simple, since the process table for the cygwin library applies only to processes which have been started by programs working under the Unix process emulation. Hopefully this short-coming can be worked around at some point in the future.

7.16 Anonymous FTP example

Configuring a service like anonymous FTP requires a certain amount of vigilance. It is a good idea to automate it and let cfengine make sure that things don't go astray. Note that we constantly ensure that the ls program used by the anonymous ftp server is a trusted program by checking it with an md5 signture of a trusted version of the program. If for some reason it should be replaced with a Trojan horse, cfagent would notice the incorrect checksum (md5) and move the bad program to ls.cf-saved and immediately replace it with the correct version without waiting for the adminstrator to act. The inform and syslog options ask for an explicit warning to be made about this copy. Here is a complete anonymous ftp setup and maintenance program for Solaris hosts.

```
control:

   actionsequence = ( directories copy editfiles files )

   # Define variables

   ftp = ( /usr/local/ftp )
   uid = ( 99 )  # ftp user
   gid = ( 99 )  # ftp group

directories:

 solaris::

   $(ftp)/pub      mode=644 owner=root group=other
   $(ftp)/etc      mode=111 owner=root group=other
   $(ftp)/dev      mode=555 owner=root group=other
```

```
   $(ftp)/usr      mode=555 owner=root group=other
   $(ftp)/usr/lib  mode=555 owner=root group=other

files:

  solaris::

   $(ftp)/etc/passwd mode=644 o=root    action=fixplain
   $(ftp)/etc/shadow mode=400 o=root    action=fixplain
   $(ftp)/pub        mode=644 owner=ftp action=fixall  recurse=inf

copy:

  solaris::

     # Make sure ls is a trusted program by copying
     # a secure location...

   /bin/ls dest=$(ftp)/usr/bin/ls
           mode=111
           owner=root
           type=checksum
           inform=true
           syslog=true

   /etc/netconfig dest=$(ftp)/etc/netconfig mode=444 o=root

   /devices/pseudo/mm@0:zero      dest=$(ftp)/dev/zero      mode=666 o=root
   /devices/pseudo/clone@0:tcp    dest=$(ftp)/dev/tcp       mode=444 o=root
   /devices/pseudo/clone@0:udp    dest=$(ftp)/dev/udp       mode=666 o=root
   /devices/pseudo/tl@0:ticotsord dest=$(ftp)/dev/ticotsord mode=666 o=root

   /usr/lib        dest=$(ftp)/usr/lib recurse=2
                   mode=444
                   owner=root
                   backup=false
                   include=ld.so*
                   include=libc.so*
                   include=libdl.so*
                   include=libmp.so*
                   include=libnsl.so*
                   include=libsocket.so*
                   include=nss_compat.so*
                   include=nss_dns.so*
                   include=nss_files.so*
                   include=nss_nis.so*
                   include=nss_nisplus.so*
                   include=nss_xfn.so*
                   include=straddr.so*

   /usr/share/lib/zoneinfo dest=$(ftp)/usr/share/lib/zoneinfo
                    mode=444 recurse=2 o=root type=binary

editfiles:

   solaris::

    #
```

```
  # Make sure that umask is right for ftpd
  # or files can be left 666 after upload!
  #

{ /etc/rc2.d/S72inetsvc

PrependIfNoSuchLine "umask 022"
}

{ $(ftp)/etc/passwd

AutoCreate
EmptyEntireFilePlease
AppendIfNoSuchLine "ftp:x:$(uid):$(gid):Anonymous FTP:$(ftp):/bin/sync"
}

{ $(ftp)/etc/group

AutoCreate
EmptyEntireFilePlease
AppendIfNoSuchLine "ftp::$(gid):"
}

{  $(ftp)/etc/shadow

AutoCreate
EmptyEntireFilePlease
AppendIfNoSuchLine "ftp:NP:6445::::::"
}

# Finally...useful for chown

{ /etc/passwd

AppendIfNoSuchLine "ftp:x:$(uid):$(gid):Anonymous FTP:$(ftp):/bin/sync"
}

{ /etc/group

AppendIfNoSuchLine "ftp::$(gid):"
}
```

7.17 WWW security

The security of the web is a slightly paradoxical business. On the one hand, we make a system for distributing files to anyone without the need for passwords, and on the other hand we are interested in limited who gets what information and who can change what. If you want web privacy you have to exclude the possibility of running untrusted CGI scripts, i.e. CGI programs which you did not write yourself since CGI programs can circumvent any server security. This is because of a fundamental weakness in the way that a WWW server works. It makes user-CGI scripts incompatible with the idea of private WWW areas.

The problem with CGI is this: in order for the httpd daemon to be able to read information to publish it, that information must be readable by the UID with which httpd runs (e.g. the www special user (you should not run with uid nobody since that can be mixed up with NFS mappings)). But CGI programs automatically run with this www UID also. Since it is not possible to restrict the actions of CGI programs which you did not write yourself, any CGI program has automatically normal file permission access to any file which the server can see. A CGI program could choose to open a restricted file circumventing the security of the daemon. In short, privacy requires a separate UID (a separate daemon and port number) or a separate server host altogether.

Provided you acknowledge this weakness, you can still use cfengine to administrate the permissions and access files on say two WWW servers from your central location. Let us imagine having a public WWW server and a private WWW server and assume that they have a common user/UID database. We begin by defining a user-ID and group-ID for the public and private services. These need to have different ID's in order to prevent the CGI trick mentioned above.

```
editfiles:

 wwwpublic::

  { $(publicdocroot)/.htaccess

  AutoCreate
  EmptyEntireFilePlease
  AppendLine "order deny,allow"
  AppendLine "deny from all"
  AppendLine "allow from all"
  }

 wwwprivate::

  { $(privatedocroot)/.htaccess

  AutoCreate
  EmptyEntireFilePlease
  AppendLine "order deny,allow"
  AppendLine "deny from all"
  AppendLine "allow .example.org"
  }
```

Your documnts should be owned by a user and group which is *not* the same as the UID/GID the daemon runs with, otherwise CGI programs and server-side emebellishments could write and destroy those files. You will also want to ensure that the files are readable by the www daemon, so a files command can be used to this end. You might want a group of people to have access to the files to modifiy their contents.

```
files:

wwwprivate::

  $(privatedocroot) mode=664 owner=priv-data group=priv-data act=fixall

wwwpublic::

  $(publicdocroot) mode=664 owner=public-data group=public-data act=fixall
```

7.18 Miscellaneous security of cfengine itself

```
control:

  SecureInput  = ( on )
```

If this is set cfengine will not read any files which are not owned by the uid running the program, or which are writable by groups or others.

7.19 Privacy (encryption)

Encryption (privacy) is not often a big deal in system administration. With the exception of the distribution of passwords and secret keys themselves, there is little or no reason to maintain any level of privacy when transferring system files (binaries for instance). If you find yourself using a tool like cfengine to transmit company secrets from one place to another you should probably book yourself into the nearest asylum for a checkup. Cfengine is not about super-secure communication, but it can be used to perform the simple job of file distribution through an encrypted link (e.g. as a NIS replacement or other password distributor). Cfengine uses the triple DES implemenation in the OpenSSL distribution (or equivalent) to provide 'good enough' privacy during remote copying.

The most important issue in system security is authentication. Without the ability to guarantee the identity of a user or of trusted information it is impossible to speak of security at all. Although services like pidentd can go some way to confirming the identity of a user, the only non-spoofable way of confirming identity is to use a shared secret — i.e. a password. A password works by demanding that two parties who want to trust one another must both know a piece of information which untrusted parties do not.

Following the second world war, the now famous pair, Julius and Ethel Rosenberg were convicted and executed for spying on the U.S. bomb project for the Soviet Union in 1953. At one point they improvised a clever password system: a cardboard Jell-O box was torn in two and one half given to a contact whom they later would need to identify. The complex edge shape and colour matching made a complex key quite impossible to forge. Our bodies use a similar method of receptor identification of molecules for immune responses as well as for smell (with some subtleties). Without matching secrets it is impossible to prove someone's identity.

To copy a file over an encrypted link, you write:

```
copy:

   source dest=destination encrypt=true server=myserver
                           trustkey=true
```

Bear in mind that the server must be a trusted host. Privacy won't help you if the data you are collecting are faulty. In order to use the encryption there must be a public/private key pair on each host. The public key must be known by both hosts. You can use the program cfkey to generate a new key file. This public key file must then be distributed. Cfagent/cfrun and cfservd can exchange keys securely over the network. This is fine, provided you trust the sources of the keys (how do you know the key is from the host/user who claims to have sent it?).

Under encrypted communications cfengine conceals the names and contents of files. Provided the private key files are private, this has the added side effect of authenticating both hosts for one another.

On the server side, you can choose whether root on a client host should have server-root's privileges to read protected files on the server. In the '`cfservd.conf`' file you make a list:

```
control:

  TrustKeysFrom = ( ip-address/series )

admit:

  /filetree *.domain.country root=myhost,yourhost

  /etc/shadow *.domain.country encrypt=true
```

In the second example, you can also restrict access to certain files to encrypted lines, i.e. demand that clients use a private connection to collect the file, in order to prevent wiretapping.

7.20 Trust and key races

Trust is the central issue in the security of any system. Public and private keys help you to trust other hosts, only after the genuine, legitimate public keys have been securely distributed to all relevant parties. Until that has happened, it is necessary to trust the identity of remote hosts. Cfengine provides trust policy options which decide whether keys should be exchanged on trust or not, when remote parties connect for the first time. If you do not want to blindly trust keys, you could arrange to exchange key files manually, e.g.

```
scp /var/cfengine/ppkeys/localhost.pub remote:/var/cfengine/ppkeys/root-IP-number.pub
```

or you could arrange to connect at a specific time, so minimize the chances of spoofer racing you to the finishing line in transferring a key for a given user at a given host.

Note that, even program like ssh which use "privileged ports" are no longer immune to spoofing. Privileged ports are ports which only the root user can bind to. The idea used to be, a connection on a privileged port must have come from a trusted user, because only someone with the root/Administrator password would be able to bind to a privileged port. Today, that idea is naive at best. Anyone can set up their own host, pull the plug on another and spoof an address or user identity – there are so many ways to attack a system that it is impossible to know with certainty to whom one is talking over the net. The only security one has is in being able to keep a secret key. However, if someone gets there before you, with a fake key, and claims to be you, the receiver cannot know better. This applies to any and all cryptographic software.

Cfengine secure copy is not based on SSL/TLS (although it shares some of the lower level libraries). SSL is not appropriate for a system administration tool, because it uses a trust model based on a third party, such as Verisign. Most adminisrators are not prepared to pay a fee to register every host on their network, with a trusted third party. Cfengine does not use the Secure Shell protocol either. The ssh protocol is not directly appropriate for a system management tool, because it provides only unilateral authentication of user to server. Cfengine authenticates these parties mutually, i.e. user to server, and server to user. Moreover, ssh requires a user to manually accept a key on trust, when the public keys are unknown to the parties, whereas cfengine works non-interactively. SSh uses the notion of binding to a trusted port, to confirm privileged user identity. Cfengine does not make this assumption.

7.21 Adaptive locks

Cfengine treats all of its operations as transactions which are locked. Locking prevents contention from competing processes and it also places reasonable limits on the execution of the program. The fact that operations are locked means that several cfengine programs can coexist without problems. Two locking parameters control the way in which operations can procure locks. The IfElapsed parameter tells operations that they can only be performed if a certain period of time has elapsed since the last time the action was performed. This is anti-spamming protection. The ExpireAfter parameter tells cfengine that no action should last more than a given length of time. This is protection against hanging sub-processes.

7.22 Spoofing

Spoofing refers to attempts to masquerade as another host when sending network transmissions. The cfservd program which can be used to transfer files or activate cfengine remotely attempts to unmask such attempts by performing double reverse lookups in the name service. This verifies by a trusted server that the socket address and the host name are really who they claim to be.

7.23 Race conditions in file copying

When copying files from a source, it is possible that something might go wrong during the operation and leave a corrupt file in place. For example, the disk might become full while copying a file. This could lead to problems. Cfengine deals with this by always copying to a new file on the destination filesystem (prefix .cfnew) and then renaming it into place, only if the transfer was successful. This ensures that there is space on the filesystem and that nothing went wrong with the network connection or the disk during copying.

7.24 size= in copy

As a further check on copying, cfengine allows you to define acceptable limits on the size of files. After all, sometimes errors might occur quite independently of anything you are doing with cfengine. Perhaps the master password file got emptied somehow, or got replaced by a binary, through some silly mistake. By checking making an estimate of the expected size of the file and adding it to the copy command, you can avoid installing a corrupt file and making a localized problem into a global one.

7.25 useshell= and owner= in shellcommands

There are dangers in starting scripts from programs which run with root privileges. Normally, shell commands are started by executing them with the help of a /bin/sh -c command. The trouble with this is that it leaves one open to a variety of attacks. One example is fooling the shell into starting foreign programs by manipulating the IFS variable to treat '/' as a separator. You can ask cfengine to start programs directly, without involving an intermediary shell, by setting the useshell variable to false. The disadvantage is that you will not be able to use shell directives such as `|` and `>` in your commands. The owner=uid directive executes shell commands as a special user, allowing you to safely run scripts without root privilege.

7.26 Firewalls

Cfengine is a useful tool for implementing, monitoring and maintaining firewalls. You can control what programs are supposed to be on the firewall and what programs are not supposed to be there. You can control file permissions, processes and a dozen other things which make up the configuration of a bastion host. By referencing important programs against a read only medium you can not only monitor host integrity but always be certain that you are never more than a cfengine execution away from correctness.

8 Summary

Cfengine is not a tool, it is an environment for managing host configuration and integrity. In this article it has only been possible to scratch the surface of what cfengine can do. To fully understand the syntax of the examples here you should read the documentation for cfengine.

The big advantage of cfengine over many other configuration schemes is that you can have *everything* in one file (or set of files). The global file is common to every host and yet it can be as general or as specific as you want it to be. You can use it as a front end for cron, and you can use its advanced features to make your hosts *converge* to a desired, correct state.

Variable Index

Concept Index

E

F

G

H

I

L

M

N

O

P

FAQ Index

Table of Contents