Recent Developments in Cfengine

Mark Burgess

Oslo University College
Cort Adelers gate 30
0254 Oslo, Norway
mark@iu.hio.no

Abstract

Cfengine is a distributed agent framework for performing policy-based network and system administration. It is in widespread use on Unix and NT systems. This paper describes recent changes to the cfengine tool-set, including architectural changes in order to facilitate anomaly detection research, public key methods, improved scheduling technology and search filters.

1 Introduction

Cfengine is an on-going research project, looking at distributed system administration. Since its inception in 1993, the cfengine tool-set has been adopted by a broad range of users from small businesses to huge organizations[2]. It is currently running on an estimated 100,000 nodes around the world. Since its inception, cfengine has developed considerably, although the basic framework and key principles have remained the same.

Cfengine falls into a class of approaches to system administration which is called policy-based configuration management. To some extent, policy-based configuration can be seen as a reaction to the inadequacies of control and monitoring software, now typified by many Simple Network Management Protocol (SNMP) implementations. Administrative schemes, employing autonomous agents, are the only administrative solutions which scale to large numbers of hosts[1], because this does not rely on the funnelling of configuration instructions through a serial queue, governed by a human. Today, the approach is used by many large organizations to manage anything from a handful to thousands of systems[2].

Over the last three years, development on cfengine has been slowed in order to perform some parallel research, attempting to unravel core issues, previously unaddressed by the research community. These include

- Scheduling ideas
- Game theoretical strategies
- Dependency resolution
- Anomaly detection

In addition to this work, it had become clear that certain functionality was lacking in order for the engine to fit into a business enterprise-style model of management:

- Enterprise level control (high level)
- Public key authentication.

In response to these issues, it was decided to make several changes to the cfengine tool-set, in order to pave the way for further progress. This paper describes the changes which have been implemented, or are being implemented.

2 Cfengine philosophy

What makes cfengine different from similar approaches to configuration management is that it embraces a stochastic model of system evolution. Rather than assuming that transitions between states of its model occur only at the instigation of an operator, or at at the behest of a protocol, cfengine imagines that changes of state occur unpredictably at any time.

The focus of cfengine, and supporting work[8], is this willingness to accept the idea of increasing random entropy of configuration through interaction. Users' social behaviour[9] is seen as a central and unignorable mixture of signals which tends to disorder the system configuration, over time. Cfengine holds to a set of principles, referred to as the *immunity model*, for seeking correctness of configuration. These embody the following features:

- Centralized policy-based specification, using an operating system independent language, which conceals implementation details.
- Distributed agent-based action, in which every host node is responsible for its own maintenance.
- Convergent semantics encourage every transaction to bring the system closer to an 'ideal' average-state, like a ball rolling into a potential well.
- Once the system has converged, action by the agent desists, or more usually, does not even start at all, when convergence was assured on a previous run of the agent.

The last two points are the most important. Most configuration agents either require a human to initiate change or rewrite the same constant configuration many times. In an analogous way to the healing of a body from sickness, cfengine's configuration approach is to always move the system closer to a 'healthy' state[1], or oppose unhealthy change: hence the name 'immunity model'. This idea shares several features with to the security model proposed in refs. [10, 11].

A healthy state is defined by reference to a local policy. When a system complies with policy, it is healthy; when it deviates, it is sick. In ref. [8] it was shown that a complete specification of policy determines an approximate configuration of a software system only approximately over persistent times. There are fundamental limits to the tolerances one can expect a system to satisfy with respect to policy compliance. A policy does not meaningfully ensure a precise configuration over microscopic intervals, even when the policy is repeatedly enforced, because every system is coupled to an environment of users, whose effect on the system is unpredictable. Fluctuations in the policy configuration can always occur over shorter times. They can be corrected, provided sufficient vigilance is maintained, but such fluctuations must inevitably occur for however brief an interval. Cfengine expects this average ideal state to be maintained only through such constant appraisal and adjustment, rather than assuming that a one-off implementation of policy suffices to place software into a persistent configured state.

3 Components of the cfengine framework

The main components of cfengine are (see table 1):

Table 1: Components in cfengine

Component	Cfengine 1.x	Cfengine 2.x
Agent	cfengine	cfagent
Server	cfd	cfservd
Scheduler	cron,cfwrap	cfexecd
Poller	cfrun	cfrun
Key Gen	cfkey	cfkey
Long term state	–	cfenvd
State grapher	–	cfenvgraph

- A central repository of policy files, which is accessible to every host in a domain.
- An active agent which executes intermittently on each host in a domain.
- A secure server which can assist with peer-level file sharing, and remote invocation, if desired.
- A passive information-gathering agent which runs on each host, assisting in the classification of host state over persistent times.
- Various supporting tools.

Cfengine employs centralized declaration of policy, with a democratic, distributed implementation. Once boot-strapped with a basic configuration, each agent decides, based on its current policy whether it wishes to import or update its policy specification from a trusted source. Cfengine is explicitly deaf to external commands, except for one generic command to re-check its state, using an existing policy. This command is protected from malicious, repetitive execution (spamming) attempts by an adaptive locking scheme[12] described below.

4 Classes and environment

The distributed nature of many software systems mixes local and remote notions freely and in a variety of ways. Setting configuration policy for such an array of software and hardware is a broad challenge, which must be addressed both at the detailed level, and at the more abstract enterprise level. Cfengine uses the idea of host classification to dissect a distributed environment into overlapping sets.

A class based decision structure is possible because a cfengine agent is run by every host on the network individually. Each host knows its own name, the type of operating system it is running and can determine whether it belongs to certain groups or not. Each host which runs a cfengine agent therefore builds up a list of its own attributes (called the classes to which the host belongs). Classes that are meaningful in the context of a particular host include:

1. The identity of a machine, including hostname, address, network.
2. The operating system and architecture of the host.
3. An abstract user-defined group to which the host belongs.
4. The result of any proposition about the system.
5. A time or date.

6. A randomly chosen strategy element.
7. The logical combination of any of the above, with AND (.), OR (|), NOT (!) and parentheses.

Given that the agent running on a host can determine the class attributes for the host, it can now pick out what it needs from the global policy, since policy criteria are also labelled with the classes to which they apply. A command or action is only executed if a given host is in the same class as the policy action in the configuration program. There is no need for formal decision structures, it is enough to label each statement with classes. At the simplest level, one has commands belonging only to a single class, say the operating system type of the hosts:

```
solaris::

   actions

sunos_5_8::

   actions
```

More complex combinations can perform an arbitrary covering of a distributed system[13], e.g.

```
AllBinaryServers.Hr22.OnTheHour.!exception_host::

   actions
```

where `AllBinaryServers` is assumed to be an abstract group, and `exception_host` is a host which is to be excluded from the rest. Classes thus form any number of overlapping sets, which cover the coordinate space of the distributed system (h, c, t), for different hosts h, with software components c, over time t. Classes sometimes become active in response to situations which conflict with policy,

Class predication allows policy to encompass many-to-one maps and one-to-many maps. Many hosts can belong to the same class, and therefore policy actions can be common to many hosts. Similarly, each host can be characterized by many different classes or attributes which label its intended state, recognizing the multiple functions of each node in the virtual community, and the distributed nature of software systems.

In cfengine 2.x, additional classes are automatically evaluated based on the state of the host, in relation to earlier times. This is accomplished by the additional *cfenvd* daemon, which continually updates a database of system averages, which characterize "normal" behaviour. The state of the system is examined and compared to the database, and the state is classified in terms of the current level of activity, as compared to an average of equivalent earlier times. e.g.

```
RootProcs_low_dev2
netbiosssn_in_low_dev2
smtp_out_high_anomalous
www_in_high_dev3
```

The first of these tells us that the number of root processes is two standard deviations below the average of past behaviour, which might be fortuitous, or might signify a problem, such as a crashed server. The WWW item tells us that the number of incoming connections is three standard deviations above average. The smtp item tells us that outgoing smtp connections are more than three standard deviations above average, perhaps signifying a mail flood. The setting of these

classes is transparent to the user, but the additional information is only visible to the privileged owner of the cfengine work-directory, where the data are cached.

Active incoming ports are also registered as "pin-portnumber", but this is mainly an experimental feature for future research. The resulting class list, obtained from exploring the environment of the system, and after parsing a configuration, looks something like this:

```
host% cfengine -p -v

[snip]

Defined Classes = ( any Thursday Hr14 Min24 Min20_25
Day19 July Yr2001 solaris examplehost 32_bit sunos_5_7
sunos_sun4u sunos_sun4u_5_7 sparc solaris2_7 129_0_0
129_0_0_10 loghost OnTheHour peaktime DayTime
examplehost_example_org longjob Setup_SSH_OK y MailHub
percent_60 RootProcs_normal_dev2 nfsd_out_low_dev2
pin-1554 pin-80 pin-21 pin-6011 pin-5308 pin-139
pin-983 pin-10 )

[snip]
```

It is not yet known how the extra environment classes will be used in practice. One obvious possibility is to limit certain heavy-weight operations (such as file tree scans) when the host is very busy, and to increase the probability of their occurrence when the host is lightly loaded. See the example in section 11. It remains to be seen how users will respond to these possibilities.

5 Aims for cfengine 2

One of the aims in stepping up a major version number is to make a clean break with some earlier practices, so that simplifications can be made with regard to installation. Getting started with cfengine version 1.x was harder than it ought to be. It was felt that the installation should be more in the manner of a plug'n'play turn-key solution, with only the policy left to write. Another ongoing aim is to keep the size of the component binaries as low as possible, while avoiding the use of shared libraries, which might introduce security weaknesses.

The dependency of cfengine 1.x on several Perl scripts has been a stumbling block for some users. These will be eliminated in cfengine 2.x, and a scheduling daemon (written in C) will replace them. This will both simplify installation and permit more adaptive scheduling of the agent, based on our latest work at Oslo University College. In cfengine 2.x, the only thing a user should have to do is to install the binaries, and fill in the configuration files. No customization should be necessary.

A greater reliance has been placed on the Berkeley Database file format, in cfengine 2.x. It is now considered a prerequisite that the system have the latest version of the Berkeley database. Use of the database format allows for a greater efficiency of storage for adaptive locks, state data, and checksum information. By using smart algorithms, the size of these databases can be kept to a minimum.

6 Work directory

In order to achieve the desired simplifications, it was decided to reserve a private work area for the cfengine tool-set. In cfengine 1.x, the administrator could

choose the locations of configuration files, locks, and logging data independently. In cfengine 2.x, this diversity has been rationalized to a single directory which defaults to `/var/cfengine` (by analogy with /var/cron)

```
/var/cfengine
/var/cfengine/bin
/var/cfengine/inputs
/var/cfengine/outputs
```

The Unix naming scheme is such a mess that it is never an easy decision to find a logical or consistent home for binaries and special workspace. Although conventions exist, they are seldom consistent across platforms. Thus, a fairly arbitrary decision had to be taken. The directory `/var/cfengine/bin` is used to cache a working version of the binaries on the local file system. The installation location `/usr/local/sbin` is not necessarily a local file system, and cannot therefore be trusted i) to be present, and ii) to be authentic on an arbitrary system.

Similarly, a trusted cache of the input files must now be maintained in the `inputs` sub-directory. When cfengine is invoked by the scheduler, it reads only from this directory. It is up to the user to keep this cache updated, on each host. This simplifies and consolidates the cfengine resources in a single place. The environment variable CFINPUTS still overrides this default location, as before, but in its absence or when called from the scheduler, this becomes the location of trusted files. A special configuration file cf.update is parsed and run before the main configuration is parsed, which is used to ensure that the currently caches policy is up-to-date.

The output directory is now a record of spooled run-reports. These are mailed to the administrator, as previously, or can be copied to another central location and viewed in an alternative browser..

Cfengine 2 separates the update of its configuration policy from the remainder of its duties, by looking for a file cf.update in /var/cfengine/input/cf.update. This file is parsed and executed before the main policy, and can therefore be used to provide a simple invariant prescription for updating policy from a trusted source.

7 Security and trust

Cfengine 1.x authenticates connections between servers either by Internet identity, or by secret keys. While this is secure for hosts in a common zone of trust, it is inflexible in a large enterprise where several zones of trust might exist.

In cfengine 2.x, RSA style, public-private key authentication is used both to validate the server connection and to validate the configuration files themselves irrespective of whether channels are encrypted. Diffie-Hellman key exchange is used to exchange a 3DES private session key. Although this does not increase the security of the transfer, it does increase the specificity of the privacy. Efforts are being made to make this mechanism compatible with the Secure Shell (using code from OpenSSH), so that only one, common set of authentication keys must be managed. Input files, and other files, may be signed with a private key in order verify authenticity.

In spite of the strong authentication used, time need not be wasted by necessarily encrypting the channel: cfengine does not insist on the use of encrypted channels. Encryption is normally only required in a few cases, for which it can be invoked explicitly. Most file transfers involve the copying of public files, and considerable efficiency savings can be won by the server, by not encrypting transfers.

Permission to invoke the agent is restricted to certain local users, and requires the invoker to obtain sufficient privilege in advance, to implement the proposed policy (no setuid mechanisms are used, for example), so the ability to validate user

names is merely a convenience. Convergence also makes restricting the identity of the invoker strictly unnecessary. If unauthorized persons somehow invoked the system, the worst they could do would be to force the system to converge to its ideal state. This presupposes, naturally that the policy itself is secure.

To simulate the immediacy of a push update, any agent can be activated (with its existing policy) by a trusted remote user, so that polling the agents requests them to pull their a new configuration, if the current policy of the agent includes automatic updates. Repetitive abuse of this feature, for mischievous ends, is defeated by an adaptive locking policy, described below.

The agent implementation described allows transparent delegation, with bottom-up override control, i.e. each host at a site elects to join a common configuration, or remain outside. No central authority can force the administrator of a host to comply with a policy, but once the decision has been made to join part or parts of such a policy, it will be fully implemented. This implies a high degree of network security, and means that potential intruders cannot force their will upon hosts by abuse of the system. In today's networks, it is impossible to absolutely force hosts to follow an external policy, if local administrators do not wish it, since ultimate control over a host is available to anyone with physical access to the machinery. In many large organizations, management is delegated to departments, who are better placed to arrange their own policy needs than a central authority. Cfengine makes this straightforward, but allows the importation of policy elements from other sources, in a flat, rather than a hierarchical model.

8 Recursion and scheduling

One of the less satisfying aspects of cfengine has been the necessity of managing the `actionsequence` list by hand. Sometimes cfengine actions trigger follow-up actions, by defining new classes based on the outcome. In cfengine 1.x, such follow-up actions could be dropped unless explicitly scheduled by the user.

Cfengine 2.x retains the idea of an action sequence, which determines the coarse ordering of actions, however it also keeps track of which rules have been executed and which have not. It then seeks to process the action sequence, by order, a maximum of three times in order to resolve any actions which were triggered by actions on previous passes. This relieves the burden and confusion of managing follow-up actions. The scheduling is transparent to the user, but can be observed in verbose mode. For example, consider a copy action which triggers a file-check operation, by the definition of a class called "signal":

```
control:

 actionsequence = ( files copy )
 AddInstallable = ( signal )

files:

  signal::

    /etc/passwd mode=0644 owner=root action=fix

copy:

 /etc/passwd dest=/etc/passwd server=sourcehost define=signal
```

In cfengine 1, the follow-up signal would not be carried out in this script, since cfengine is done with `files` before `copy`, thus the file action which becomes activated by the copy would be dropped, unless the user had the presence of mind to add `files` a second time to the action sequence, or changed the ordering. In cfengine 2.x, cfengine will iterate the actionsequence until such dependencies are resolved.

9 Strategies

In the theory of games, the concept of a strategy is a set of actions taken by one of the players in order to win the competition. Often a player can benefit by using a variety of tactics, in a *mixed strategy*, so as to either confuse the opponent, or simply seek a compromise between multiple aims.

In cfengine 2.x, the concept of a strategy is like that of a mixed game strategy. A strategy is a set of classes, each of which is used to label a strategy for configuration management. A strategy definition looks like this:

```
strategies:

  { spread_load

  percent_10: "1"   # These classes get defined in these ratios
  percent_30: "3"
  percent_60: "6"
  }
```

This declaration defines one and only one of the set of classes `percent_10`, `percent_30`, `percent_60`, on each invocation of the agent `cfagent`. The probability that the class will be defined is determined by the quoted integers. The sum of the integers in a strategy is used to define a total which divides each of the integers, thus leading to a fraction between zero and one, which is the probability with which the classes should be defined. A Monte Carlo (metropolis) algorithm is then used to select one of the classes at random.

The name of the strategy is used only as a convenience; it is not referred to anywhere else, but it does identify the elements as being part of an entity. The identifier can be used to modify the strategy with class-based exceptions. It is possible, for instance, to insert an additional element on a special host,

```
strategies:

  # ...

 SpecialHost::

  { spread_load

  dominant_strategy:: "10"
  }
```

though the addition of this class would cause the percentages to be recalculated, so that the percent names above would cease to be correct (the total would then be $1+3+6+10=20$, giving fractions $1/20, 3/20, 6/20, 10/20$, or 5%,15%,30%,50%).

Once a strategy element has been selected, at random, the element is then treated as an ordinary class. For example:

```
processes:

 Nameservers.percent_10::       # memory problems named

  "named" signal=term restart "/local/sbin/named -u dns"
```

This rule kills and restarts the *named*, nameserver daemon ten percent of the time. This turns out to be an effective strategy against the daemon crashing, due to an internal error (presumably a memory leak). To randomize the occurrence of a particular action in time, one could try a strategy like the following:

```
tidy:

  percent_10.Hr16::

      tidy rules.....

  percent_30.Hr20::

      tidy rules.....
```

This would ensure that tidying was performed 10 percent of the time, at 4 p.m., while thirty percent of the time at 8 p.m. (but possibly both). This application of strategies can be used for load spreading, for instance. Randomization of time can be applied to backups, integrity checks, checks on connections, log rotation, and so on.

Other applications could include randomizing links to actual binaries, to thwart intrusion attempts on systems which require high security and have high risk. In the configuration policy were secret, one could also randomize the contents of key variables for security purposes: location of backup host, choice of filesystem to check, location of key programs (e.g. link to alternative versions e.g. passwd or ps, which are often the object of Trojans). This is essentially a temporary diversion based on "security through obscurity"; however, since it is dynamical and random, rather than fixed, it is possible to win some effect from this game-theoretically. This kind of tactic might be sufficient to delay the efficacy of an intrusion for long enough to detect it and find a more permanent fix.

10 Search Filters

Large scale configuration control and regulation often requires searching through files or processes for very specific faults. Tools for performing this kind of search have been lacking. On Unix systems, advanced file searches can be accomplished with the `find` command, but the syntax of this command is clumsy and difficult to incorporate into complex rules.

In cfengine 1.6, the concept of a filter was introduced in order to pick out specific files or processes in a search. A filter is a way of selecting or pruning during a search over files or processes. Since filter rules could apply to several objects, cfengine allows you to define filter conditions as separate objects to be applied in different contexts. In a sense, filters classify the files found during a search, in much the same way that class attributes classify hosts in a domain.

Filter objects can be used in `copy`, `editfiles`, `files`, `tidy` and `processes`. In most cases one writes

```
.. filter=filteralias
```

in the appropriate command. The exception is `editfiles`, where the syntax is

```
{
..
Filter "filteralias"
..
}
```

Example:

```
files:

 /tmp filter=testfilteralias action=alert r=inf
```

Filters are defined in a separate section. Filters for files and processes are defined together. They differ only in the criteria they contain.

10.1 File filters

File filters contain the following propositions:

- **Owner:** and **Group:** can use numerical user id's or names, or `"none"` for users or groups which are undefined in the system passwd/group file.
- **Mode:** applies only to file objects. It shares syntax with the `mode=` strings in the files command. This test returns true if the bits which are specified as 'should be set' are indeed set, and those which are specified as 'should not be set' are not set.
- **Atime:,Ctime:,Mtime:** apply only to file objects. These specify ranges From and To. If the file's time stamps lie in the specified range, this returns true. Times are specified by a six component vector

  ```
  (year,month,day,hour,minutes,seconds)
  ```

  This may be evaluated as two functions: **date()** or **tminus()** which give absolute times and times relative to the current time respectively. In addition, the words **now** and **inf** may be used. e.g.

  ```
  FromCtime: "date(2000,1,1,0,0,0)"  # absolute date
  ToCtime:   "now"

  FromMtime: "tminus(1,0,0,2,30,0)"  # relative "ago" from now
  ToMtime:   "inf"                   # end of time
  ```

- **Type:** applies only to file objects. It may be a list of file types which are to be matched. The list should be separated by the OR symbol (`\|`), since these types are mutually exclusive. The possible values are currently

  ```
  reg|link|dir|socket|fifo|door|char|block
  ```

  The 'door' category applies only to Solaris.

- **ExecRegex:** matches the parenthetic regular expression against the output of the specified command, in the form "Command (test-string)".

- **NameRegex:** matches the name of the file with a regular expression.
- **IsSymLinkTo:** applies only when the file object `$(this)` is a symbolic link. It is true if the regular expression matches the contents of the link.
- **ExecProgram**: matches if the command returns successfully (with return code 0). Note that this feature introduces an implicit dependency on the command being called. This might be exploitable as a security weakness by advanced intruders.
- **Result:** specifies the way in which the above elements are combined into a single filter.

10.2 Process filters

Process filters may contain these propositions:

- **PID:** process ID (quoted regex)
- **PPID:** parent process ID (quoted regex)
- **PGID:** process group ID (quoted regex)
- **RSize:** resident size (quoted regex)
- **VSize:** virtual memory size (quoted regex)
- **Status:** status (quoted regex)
- **Command:** CMD or COMMAND fields (quoted regex)
- **(From/To)TTime:** Total elapsed time in TIME field (accumulated time)
- **(From/To)STime:** Starting time for process in STIME or START field (accumulated time)
- **TTY:** terminal type, or none (quoted regex)
- **Priority:** PRI or NI field (quoted regex)
- **Threads:** NLWP field for SVR4 (quoted regex)
- **Result:** logical combination of above returned by filter (quoted regex)

Examples: processes started between 18th Nov 2000 and now.

```
{ filteralias

FromSTime: "date(2000,11,18,0,0,0)"
ToSTime:   "now"
}
```

All processes which have accumulated between 1 and 20 hours of CPU time.

```
{ filteralias

FromTTime:  "accumulated(0,0,0,1,0,0)"
ToTTime:    "accumulated(0,0,0,20,0,0)"
}
```

10.3 Complete examples

Here is an example filter which searches for all files which are either directories or links, and additionally any kind of file owned by mark, in group cfengine.

```
control:

 actionsequence = ( files )

files:

 /tmp       filter=testfilteralias action=alert r=inf
 /cfengine  filter=testfilteralias action=fixall r=inf mode=644

filters:

  { testfilteralias

  Owner:     "mark"
  Group:     "cfengine"
  Type:      "dir|link"

  Result:    "Type|(Owner.Group)"
  }
```

To find all ELF executables using data from the Unix `file` command, one could use the following code. Note that this approach of running subservient shell commands takes a long time if used indiscriminately.

```
control:

 actionsequence = ( files )

files:

 /tmp       filter=testfilteralias action=alert r=inf
 /cfengine  filter=testfilteralias action=fixall r=inf mode=644

filters:

  { testfilteralias

  ExecRegex: "/bin/file (.*ELF.*)"

  Result: "ExecRegex"
  }
```

Here is an example which warns of any process coupled to a terminal (pty) started in November (month 11):

```
control:

 actionsequence = ( processes )

filters:
```

```
  { filteralias
  FromSTime: "date(2000,11,0,0,0,0)"
  ToSTime:   "date(2000,11,30,0,0,0)"
  TTY: ".*pt.*"
  Result: "TTY.STime"
  }

processes:

 "." filter=filteralias action=warn
```

To search for users who have linked their history files to `/dev/null` (a common trick when they are up to no good), one may use the following rule:

```
file:

 /homedirs recurse=3 action=alert filter=historyfilter


filters:

  { historyfilter

  NameRegex:   ".*history"
  IsSymLinkTo: "/dev/null"
  Result:      "IsSymLinkTo.NameRegex"
  DefineClasses: "historyalert"
  }
```

Some further examples of file filters:

```
Filters:

  { filteralias1

  Owner:     "mark|cell|motd"
  Group:     "ecg|mark"
  Mode:      "700"

  FromCtime: "date(2000,1,1,0,0,0)"     # absolute date
  ToCtime:   "now"

  FromMtime: "tminus(1,0,0,2,30,0)"     # relative "ago" from now
  ToMtime:   "inf"                      # end of time

  FromAtime: "date(1997,2,22,0,0,0)"
  ToAtime:   "inf"

  FromSize:  "10000"                    # File size interval
  ToSize:    "10mb"

  ExecRegex: "/usr/bin/file (.*ascii.*)" # Use "file" command

  Type:      "dir|link"

  NameRegex: ".*.asc"                   # regex matching file name

  IsSymLinkTo: ".*null"                 # True if file links to name-regex
```

```
 Result:    "Type"                     # Result to be returned

 }

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

 { testfilteralias2

 ExecProgram: "/bin/ls $(this)"       # True if the program returns true;
                                      # $(this) is the current object in
 }                                    # the search

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

 { testfilteralias3

 Owner: "mark"
 }
```

Filters outcomes are evaluated like classes. In fact, the filtering works by evaluating the class attributes for each file.

11 Anomaly Research

There is no system available in the world today which can claim to detect and classify the functioning state of a computer system. Cfengine does not attempt to provide a "product" solution to this problem; rather it incorporates a framework, based on the current state of knowledge, for continuing research into this issue.

In version 2.x of cfengine, an extra daemon (cfenvd) is used to collect statistical data about the recent history of each host (approximately the past two months), and classify it in a way that can be utilized by the cfengine agent. Data are are gradually aged so that older values count less. One must have the Berkeley database installed in order to use this. The cf-environment daemon is meant be trivial to use. The current long-term data recorded by the daemon are: number of users, number of root processes, number of non-root processes, percentage disk full for root disk, number of incoming and outgoing sockets for netbiosns, netbiosdgm, netbiosssn, irc, cfengine, nfsd, smtp, www, ftp, ssh and telnet. These data have been studied previously, and their behaviour is relatively well understood. In future versions, it is expected to extend this repertoire, as more research is done.

The use of the daemon will not be reliable until about six to eight weeks after installing and running it, since a suitable training period is required to build up enough data for stable characterization. The daemon automatically adapts to the changing conditions, but has a built-in inertia which prevents anomalous signals from being given too much credence. Persistent changes will gradually change the 'normal state' of the host over an interval of a few weeks. Unlike some systems, cfengine's training period never ends. It regards normal behaviour as a relative concept, which has more to do with local stability than global constancy.

The cfenvd daemon does not have to be run with root privileges, but it must be able to write to a database. The database records one week's worth of data which are iteratively updated in order to give an approximate decaying average based on two months worth of data. The size of the database is approximately 2MB. Measurements are taken every five minutes (approximately). This interval is based on auto-correlation times measured for networked hosts in practice.

Cfenvd sets a number of classes in cfengine which describe the current state of the host in relation to its recent history. The classes describe whether a parameter is above or below its average value, and how far from the average the current value is, in units of the standard-deviation (see above). This information could be utilized to arrange for particularly resource-intensive maintenance to be delayed until the expected activity was low.

The cfenvgraph command can use used to dump a graph of averages for visual inspection of the normal state database. The format of the file is

$$t, y_1, y_2, y_3 ...$$

which can be viewed using *gnuplot* or *xgmr* or other graphical plotting program. This would allow the policy-maker to see what is likely to be a good time for such work (say 06:00 hours), and then use this time for the job, unless an anomalous load is detected. For instance, since significant server activity could place a significant load on a host

```
copy:

Hr06.!(www_in_high_dev3||www_in_high_anomalous)::

 /www-user-database dest=/www-backup
```

The concept of load average is not used here, primarily because it is ambiguous and too unspecific.

One way of understanding cfengine's behaviour in relation to environment and policy is as a Markov fluctuation model of change[18], seeking an equilibrium configuration. Recently, a transaction manager was introduced into cfengine, to cope with the need for long-term, persistent memory (so-called non-Markovian variables). In the original model, only first order Markov processes were available; everything about the system had to be deduced from the environment upon each new invocation. The only exception was the use of adaptive locks, which were used to regulate repetition. That approach has succeeded in solving many problems, but it lacks the ability to track long-term changes to the system, such as seasonal variations and changing patterns of usage. In version 2 of cfengine, classes can be based on long term memory which degrades over a period of approximately a month (the time over which computers may be regarded as being statistically stable[9]). This allows for a more refined statistical sense of anomaly.

The immunity analogy is also useful here. Immunological memory is like a stack of previously combatted problems, which works like an ordered list of changing priorities. This enables repeatedly actual problems to be dealt with more quickly than otherwise. It represents a change in biological policy toward threat. The same problem arises in configuration management, in the face of unpredictable faults or attacks.

The transaction manager collects data, including statistical samples of system performance variables, and an environment daemon collates and classifies these data into regular cfengine classes, which can be used to activate actions to counter emerging problems, or even modify strategy. Experience with such mechanisms is currently limited, but they are a logical generalization of the first order classifications, which allow for more complete knowledge of competition in the face of environmental complexity. The data in such a database, are a natural candidate for analysis, to be fed back into a refinement of policy.

12 Summary

The recent changes in cfengine can be viewed as a rationalization of the creeping featurism of earlier versions, together with some additional architectural changes and game theoretical features. The architecture changes will permit future experimental features to be explored, in a harmless fashion to users, and allow a gradual adoption of new possibilities.

Acknowledgment

I wish to express my gratitude to the many users of cfengine who have helped to improve the details of implementation over the years, while tolerating my insistence on adherence to core principles. Cfengine is available as free, open source software from ref. [3]. I am grateful for Alva Couch for a critical reading of the manuscript.

References

[1] M. Burgess. Computer immunology. *Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.

[2] M. Burgess. Evaluation of cfengine's immunity model of system maintenance. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, 2000.

[3] M. Burgess. Cfengine www site. *http://www.iu.hio.no/cfengine.*

[4] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.

[5] B. Hagemark and K. Zadeck. Site: a language and system for configuring many computers as one computer site. *Proceedings of the Workshop on Large Installation Systems Administration III (USENIX Association: Berkeley, CA, 1989)*, page 1, 1989.

[6] D. Marriott and M. Sloman. Implementation of a management agent for interpreting obligation policy. *Implementation of a management agent for interpreting obligation policy*, IFIP/IEEE 7th international workshop on distributed systems operations and management (DSOM), 1996.

[7] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, **16**:1293, 1990.

[8] M. Burgess. On the theory of system administration. *Submitted to J. ACM.*, 2000.

[9] M. Burgess, H. Haugerud, T. Reitan, and S. Straumsnes. Measuring host normality. *ACM / Transactions on Computing Systems (submitted)*, 2001.

[10] P.D'haeseleer, Forrest, and P. Helman. An immunological approach to change detection: algorithms, analysis, and implications. *In Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy (1996).*

[11] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. *New Security Paradigms Workshop*, September 1997.

[12] M. Burgess and D. Skipitaris. Adaptive locks for frequently scheduled tasks with unpredictable runtimes. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 113, 1997.

[13] D.E. Comer and L.L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3:51, 1989.

[14] N. Damianou, N. Dulay, E.C. Lupu, and M. Sloman. Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.

[15] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software practice and experience*, 27:1083, 1997.

[16] A. Couch. An expectant chat about script maturity. *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV) (USENIX Association: Berkeley, CA)*, page 15, 2000.

[17] S.C. Cheung and J. Kramer. An integrated method for effective behaviour analysis of distributed systems. *Proc. of 16th IEEE Int. Conf. on Software Engineering (ICSE-16), Sorrento*, page 309, 1994.

[18] G.R. Grimmett and D.R. Stirzaker. *Probability and random processes*. Oxford scientific publications, Oxford, 1982.