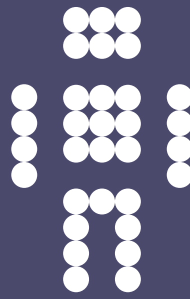
CFEngine



Security Scanning and Tripwires with CFEngine

A CFEngine Special Topics Handbook

CFEngine AS

CFEngine has sophisticated functionality for scanning hosts to find anomalous content, for looking through log messages and detecting unauthorized file changes. This can form the basis of a host based intrusion shield, either alone or in conjunction with other tools.

This document describes how to scan systems for potential security incidents and vulnerabilities, and view reports across your system using the Mission Portal.

Copyright © October 2011 CFEngine AS

Table of Contents

1 File scanning 1
1.1 File change detection: tripwires 1
1.2 Example change management 2
1.3 file_change.log 4
1.4 File change reports in Nova/Enterprise 4
1.5 Tamperproof data 6
1.6 Change detection bundle summary 7
1.7 Reversion of file changes from a trusted source 9

2 Process and network scanning 11
2.1 Watching processes 11
2.2 Watching other system variables 12
2.3 Threshold monitoring 15
2.4 Port monitoring 15

3 Log scanning 17
3.1 Scanning log files for patterns 17
3.2 Scanning syslog for FTP statistics 18
3.3 Scanning DNS logs for query statistics 19
3.4 Scanning syslog for email statistics 20
3.5 Scanning syslog for email milter failures 21
3.6 Scanning syslog for breakin attempts 22

4 Intrusion detection 25

1 File scanning

CFEngine has sophisticated functionality for scanning hosts to find anomalous content, for looking through log messages and detecting unauthorized file changes. This can form the basis of a host based intrusion shield, either alone or in conjunction with other tools.

This document describes how to scan systems for potential security incidents and vulnerabilities, and view reports across your system using the Mission Portal.

1.1 File change detection: tripwires

File change monitoring is about detecting when file information on a computer system changes. You might or might not know that files are going to change. Expected changes are not usually a problem, but unexpected change can be problematic or even sinister.

The bulk of information on a computer is its file data. Change detection for filesystems uses a technique made famous in the original open source program Tripwire, which collects a snapshot of the system in the form of a database of file checksums (cryptographic hashes) and then periodically rechecks the system against this database to see what has changed. Using cryptographic hashes is an efficient way of detecting change as it reduces file contents to a unique number, just a few bytes long, which can be stored for later comparison to detect change.

If as much as a single bit of information changes, the file hash will change by a noticable amount. This is a very simple (even simplistic) view of change, but it is effective at warning about potential incursions to the system.

A cryptographic hash (also called a digest) is an algorithm that reads (digests) a file and computes a single number (the hash value) based on its contents. If so much as a single bit in the file changes then the value of the hash will change. You can compute hash values manually, for example:

```
host$ openssl md5 /etc/passwd
MD5(/etc/passwd)= 1fbd82252c441d0e9539f8f7271ec2fe
```

There are several kinds of hash function. The most common ones are MD5 and SHA1. Recently both of these algorithms have been superceded by the newer SHA2 set.

Note that the FIPS 140-2 US government standard for encryption does not recognize the MD5 hash algorithm. The default algorithm for enterprise grade CFEngine is now SHA256.

To use hash based change detection we use '`files`' promises and the `changes` feature; first we ask CFEengine to compute file hashes for specified files and enter them into a database. Then, the same promise on subsequent runs will re-collect the data and compare the result to what has been stored in the database.

Here is a simple CFEngine promise that checks for changes in '`/usr/local`':

```
files:
```

```
  "/usr/local"
     changes      => detect_all_change,
     depth_search => recurse("inf");
```

This example uses the standard library template '`detect_all_change`'.

The first time this promise is kept, CFEngine collects data and treats all files as *unchanged*. It builds a database of the checksums. The next time the rule is checked, cfagent recomputes the checksums and compares the new values to the 'reference' values stored in the database. If no change has occurred, the two should match. If they differ, then the file as changed and a warning is issued either on the command line (if you are testing manually) or by email.

```
cf3: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
cf3: SECURITY ALERT: Checksum (sha256) for /etc/passwd changed!
cf3: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

This message is designed to be visible. If you do not want the embracing rows of '!' characters, then this control directive turns them off:

```
body agent control
{
exclamation => "false";
}
```

1.2 Example change management

Try the following complete example.

```
body common control
{
bundlesequence => { "test_change" };
inputs => { "LapTop/cfengine/copbl/cfengine_stdlib.cf" };
}

bundle agent test_change
{
files:

  "/tmp"
       changes => detect_all_change,
   depth_search => recurse("inf");
}
```

This example shows how we use standard library templates to scan a directory and (recursively) all of its sub-directories and their contents.

The first time we run this, we get a lot of messages about new files being detected. This learns and defines the baseline for future comparisons:

```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/test.cf -K

 !! File /tmp/.X0-lock was not in sha512 database - new file found
I: Made in version 'not specified' of '/home/a10004/test.cf' near line 14
 !! File /tmp/pulse-5weilfdGWBDj/pid was not in sha512 database - new file found
I: Made in version 'not specified' of '/home/a10004/test.cf' near line 14
```

Next we can create a new file in the directory:

```
host$ touch /tmp/blablabla
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/test.cf -K

 !! File /tmp/blablabla was not in sha512 database - new file found
I: Made in version 'not specified' of '/home/a10004/test.cf' near line 14
```

Next we edit the contents of the file

```
host$ echo sldjfkdsf > /tmp/blablabla
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/test.cf -K

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ALERT: Hash (sha512) for /tmp/blablabla changed!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 -> Updating hash for /tmp/blablabla to SHA=63fc10a0c57dc8cbd2c259b4d0bb81e1b4e5cf23f1fdc8b8
I: Made in version 'not specified' of '/home/a10004/test.cf' near line 14

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ALERT: Last modified time for /tmp/blablabla changed Thu Oct 20 08:46:58 2011 -> Thu Oct 20
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Note the different messages here. The first message notes that the contents of the file have changed. The second indicates that the database has been updated with the new hash of the file so that this message will not be regenerated on every subsequent run of CFEngine. Finally, it notes that the modification time on the file changed. These messages reflect the default settings in the 'detect_all_change' template (see below).

Finally, if we remove a file from a monitored directory, we see the following:

```
host$ rm /tmp/blablabla
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/test.cf -K

ALERT: /tmp/blablabla file no longer exists!
I: Made in version 'not specified' of '/home/a10004/test.cf' near line 14
```

In this example, we used the standard library template 'detect_all_change', which is defined as follows:

```
body changes detect_all_change
{
hash           => "best";
report_changes => "all";
update_hashes  => "yes";
}
```

This is the most exacting option, but it costs CPU time to compute two different hashes for cross-referencing as well as for reading disk files and it costs disk space in storing both content

hashes and additional file attribute information. Moreover, it resets its learned database each time a change is made.

There are several alternatives. To save CPU and disk, we can just monitor content:

```
body changes detect_content
{
hash           => "md5";
report_changes => "content";
update_hashes  => "yes";
}
```

To prevent the database being updated, we can use:

```
body changes noupdate
{
hash           => "sha256";
report_changes => "content";
update_hashes  => "no";
}
```

Finally, commercially supported versions of CFEngine will support detection of file difference reports in the Mission Portal, if enabled in policy:

```
body changes diff # Generates diff report (Nova and above)
{
hash           => "sha256";
report_changes => "content";
report_diffs   => "true";
update_hashes  => "yes";
}
```

1.3 file_change.log

The file file_hash_event_history contains a separate text log of file changes.

```
host$ sudo more /var/cfengine/state/file_change.log
[sudo] password for you:
1308904847,/etc/passwd
1308904847,/etc/shadow
```

The first column is a time stamp. If you are using one of the commercial versions of CFEngine, you can see a more user friendly report for the entire enterprise or for a single host in the next section.

1.4 File change reports in Nova/Enterprise

In the commercially supported editions of CFEngine, users have access to a number of reports about file changes. A report of file changes and the times at which the changes were detected is available in the file change report:

NOVA MISSION PORTAL

Hello admin | logout Online users: 0

MISSION PORTAL > ENGINEERING > COMPLIANT HOSTS > HOST > REPORT

Search in knowledge map

File change diffs for cf448lin.test.cfengine.com

PDF Select host Select report

Save this search

New search

Total results found : 3

HOST	FILE	CHANGE DETECTED AT	CHANGE ADDED(+), DELETED(-), LINE NO, CONTENT
cf448lin.test.cfengine.com	/tmp/file_change_test	October 19, 2011 16:57:29 (GMT+2)	- 3 file change test line #3 + 3 editing line #3
cf448lin.test.cfengine.com	/tmp/file_change_test	October 19, 2011 16:54:32 (GMT+2)	+ 3 file change test line #3
cf448lin.test.cfengine.com	/tmp/file_change_test	October 19, 2011 16:53:10 (GMT+2)	+ 2 file change test line #2

First < 1 > Last

20 Rows/Page

CFEngine Nova 2.1.0 running on CFEngine Core 3.3.0a1.r3205

Copyright © 2011 CFEngine AS - All rights reserved. (License information)

CFEngine does not record the actual changes unless you mention files explicitly by name, and the report is mostly useful if the file is a text file. In that case, you will see a change report:

NOVA MISSION PORTAL

Hello admin | logout Online users: 1

MISSION PORTAL > ENGINEERING > COMPLIANT HOSTS > HOST > REPORT

Search in knowledge map

File change log for cf448lin.test.cfengine.com

PDF Select host Select report

Save this search

New search

Total results found: 2025

HOST	FILE	CHANGE DETECTED AT	NOTE
cf448lin.test.cfengine.com	/tmp/file_change_test	October 19, 2011 16:57:29 (GMT+2)	notes √
cf448lin.test.cfengine.com	/tmp/file_change_test	October 19, 2011 16:54:32 (GMT+2)	notes
cf448lin.test.cfengine.com	/tmp/file_change_test	October 19, 2011 16:53:10 (GMT+2)	notes
cf448lin.test.cfengine.com	!! File /tmp/file_change_test was not in sha256 database - new file found	October 19, 2011 16:52:11 (GMT+2)	notes
cf448lin.test.cfengine.com	!! File /usr/include/crack.h was not in sha512 database - new file found	October 19, 2011 16:49:22 (GMT+2)	notes
cf448lin.test.cfengine.com	!! File /usr/include/fw_context.h was not in sha512 database - new file found	October 19, 2011 16:49:22 (GMT+2)	notes
cf448lin.test.cfengine.com	!! File /usr/include/initreq.h was not in sha512 database - new file found	October 19, 2011 16:49:22 (GMT+2)	notes
cf448lin.test.cfengine.com	!! File /usr/include/iscsi_list.h was not in sha512 database - new file found	October 19, 2011 16:49:22 (GMT+2)	notes

1.5 Tamperproof data

Message digests are supposed to be unbreakable, tamperproof technologies, but of course everything can be broken by a sufficiently determined attacker. Suppose someone wanted to edit a file and alter the cfengine checksum database to cover their tracks. If they had broken into your system, this is potentially doable, though not easy. How can we detect whether this has happened or not?

A simple solution to this is to exploit the fact that CFEngine is a distributed system, and to make neighbouring hosts work together to watch over one another. We can use the same idea for change detection, but now remotely across networked hosts: we use another checksum-based operation to copy the database to a completely different host. By using a copy operation based on a checksum value, we can also remotely detect a change in the checksum database itself.

```
bundle agent neighbourhood_watch
{
vars:

  "neighbours" slist => peers("/var/cfengine/inputs/hostlist","#.*",4),
             comment => "Get my neighbours from a list of all hosts";

files:

  # Redundant cross monitoring .......................................

  "$(sys.workdir)/nw/$(neighbours)_checksum_digests.db"

     comment => "Watch our peers remote hash tables!",
    copy_from =>
           remote_cp("$(sys.workdir)/checksum_digests.db",$(neighbours)),
  depends_on => { "grant_hash_tables" };

  # Define the actual children to watch over ........................

   "/usr/bin"
      comment     => "Watch over the system binaries",
      changes      => detect_all_change,
      depth_search => recurse("inf"),
      action       => measure;

}
```

We must have a file ‘hostlist’ containing a list of all participating hosts; it is best to list IP addresses, one per line, but host names will also do. This works as follows.

Each host extracts from the host lists a list of (in this case three) neighbours, since we have set a neighbourhood size of 4. The CFEngine built-in function ‘peers()’ can be used for this. Each host in the network takes on the responsibility to watch over its neighbours by

promising to copy the change databases for its neighbours only if they have changed. Any changes are then logged as alerts.

The copy rule attempts to copy the database to some file in a safekeeping directory `$(sys.workdir)/nw`. In theory, all four neigbours should signal this change. If an attacker had detailed knowledge of the system, he or she might be able to subvert one of these before the change was detected, but it is unlikely that all four could be covered up. At any rate, this approach maximizes the chances of change detection.

Finally, in order to make this copy, you must, of course, grant access to the database in by granting access to the database in a bundle of server promises:

```
bundle server access_rules()
{
vars:

  # List here the IP masks that we grant access to on the server

    "acl" slist => {
                   "$(sys.policy_hub)/24",
                   "128.39.89.233",
                   "2001:700:700:3.*"
                   },
       comment => "Define an acl for the machines to be granted accesses",
        handle => "common_def_vars_acl";

access:

  "/var/cfengine/checksum_digests.tcdb"

     handle => "grant_hash_tables",
    admit   => { @(acl) },
    maproot => { @(acl) };
```

Let us now consider what happens if an attacker changes a file an edits the checksum database. Each of the four hosts that has been designated a neighbour will attempt to update their own copy of the database. If the database has been tampered with, they will detect a change in the hashes of the remote copy versus the original. The file will therefore be copied.

It is not a big problem that others have a copy of your checksum database. They cannot see the contents of your files from this. A potentially greater problem is that this configuration will unleash an avalanche of messages if a change is detected. This does make messages visible however.

1.6 Change detection bundle summary

A typical promise structure for doing change management.

```
bundle agent change_management
{
vars:

  "watch_files" slist =>  {
                          "/etc/passwd",
                          "/etc/shadow",
                          "/etc/group",
                          "/etc/services"
                          };

  "watch_dirs" slist =>   {
                          "/usr"
                          };

  "neighbours"   slist => peers("/var/cfengine/inputs/hostlist","#.*",4),
               comment => "Partition the network into groups";

files:
   "$(watch_dirs)"
      comment      => "Change detection on the directories",
      changes      => detect_all_change,
      depth_search => recurse("inf");

   "$(watch_files)"
      comment      => "Change detection on important files",
      changes      => diff;  # diff_noupdate

  # Redundant cross monitoring ... neighbourhood watch

  "$(sys.workdir)/nw/$(neighbours)_checksum_digests.db"

     comment => "Watching our peers remote hashes  - cross check",
   copy_from =>
           remote_dcp("$(sys.workdir)/checksum_digests.db","$(neighbours)"),
  depends_on => { "grant_hash_tables" },
      action => neighbourwatch("File changes observed on $(neighbours)");
}

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

body action neighbourwatch(s)
{
ifelapsed    => "30";
log_string   => "$(s)";
log_repaired => "stdout";
}
```

Note that the ‘remote_dcp’ template uses a digest-comparison when comparing the files.

With this promise in place, you will not merely be warned about file changes, but you will immediately remediate the erroneous file change. By default, CFEngine backs up files when they are replaced by appending a suffix ‘.cfbackup’, or timestamp. If you want to go and look at the erroneous file for forensic evaluation, this backup contains the evidence of changes.

1.7 Reversion of file changes from a trusted source

Detecting changes to files is only half of battle against unauthorized change. In most cases you will also want to revert back to a trusted version. A recommended approach here is to maintain a trusted repository of source files, and to ask CFEngine to compare the current version of the file to master version using a cryptographic hash.

Suppose we want to manage a particular file (of any type or for any purpose) called ‘fileX’, which is to be located in some directory ‘/destination’ of a host. We arrange for some other host (that is considered ‘secure’) to have a trusted version the file in some directory called ‘/mastersources’[1]. A promise that synchronized the

```
"/destination/fileX"

   comment => "Copy fileX from a trusted source",
 copy_from =>
         remote_dcp("/mastersources/fileX","masterhost"),
depends_on => { "grant_fileX" },
    action => neighbourwatch("File changes observed on fileX");
```

We assume here that access to the masterfile has been granted to the host.

[1] Perhaps this master source has its own version control system for tracking intended changes in the master versions; we shall not discuss that here.

2 Process and network scanning

Looking at file changes is a very static view of system change – basically you assume that the system should not change from some initial snaphot, or at the very least you warn about every change. However, many aspects of a system change all the time, e.g. the network connections to and from the host, the processes running on the system.

The system might attain some kind of *statistical* normalcy, which can be learnt over time, but it that is based on a kind of equilibrium with its environment, not of locking down the system completely.

CFEngine's `cf-montord` has the ability to learn the trends and behaviour of any countable or measurable value on the system. Over time, it employs machine-learning methods from artificial intelligence to build a normalcy profile for the system. Any significant deviations from these profiles can be reported and responded to in policy.

In the Special Topics Guide *Monitoring with CFEngine*, you will find more information about about how to use CFEngine's monitoring daemon `cf-monitord` to watch over changes to the system that are dynamical.

2.1 Watching processes

CFEngine does not encourage the watching of systems without repair, i.e. simply informing humans about changes without fixing problems. After all, if you have a chance to repair something that is already encoded into policy, why wouldn't you simply do it? However, in certain mission critical environments you might want a specific alert about changes even though a repair was made. If certain conditions are not supposed to happen, it is good to know that they did.

In the following example, we look for a number of very specific processes in the process table, using our own promise-bdy template. Then we use a standard library template `check_range` to set policy for an acceptable range of such processes on the system.

This combines two kinds of checks: how many of a particular process is acceptable at any given moment, and the precise parameters satsified: in this case, the amount of memory used by the process.

```
bundle agent count_important_processes
{
processes:

 ".*"

    process_select  => my_proc_finder("myprocess"),
    process_count   => check_range("myprocess",1,10);
}

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

body process_select my_proc_finder(p)

{
process_owner  => { "root", "bin" };
command        => "$(p)";
pid            => "100,199";
vsize          => "0,1000";
process_result => "command.(process_owner|vsize)";
}
```

This example therefore searches for processes called '`myprocess`' running as root or bin that use between 0 and 1000 KB of virtual memory and have a process ID between 100 and 199. Although this example is a little contrived, it shows how different criteria can be combined to watch for very specific promises.

2.2 Watching other system variables

When `cf-monitord` is running, it collects information about a running processes and network connections to well-known ports. There is a list of standard ports and system attributes examined by CFEngine, and you can extend this using `measurements` promises.

> In the Special Topics Guide *Monitoring with CFEngine*, you will find more information about about how to use CFEngine's monitoring daemon `cf-monitord` to watch over changes to the system that are dynamical.

The information learnt by the monitoring daemon is stored in an embedded database so that CFEngine can learn the normal behaviour of the system. It does this by gathering statistics using a very lightweight, smart algorithm that avoids storing large amounts of data. The result is that every monitored value can be characterized by an expectation value and an estimate of the standard-deviation of the values measured.

CFEngine is smart enough to realize that what us normal at one time of day is not necessarily normal all the time, so it learns what is normal within each five minute interval of

the days of the week. Based on this published model, it is possible to detect anomalies in behaviour quite accurately, provided enough data are available.

Looking for statistical anomalies on little-used hosts is a waste of time. If it is normal that nothing happens, then everything that happens in anomalous.

You do not need to provide any special configuration in the monitor daemon to be able to use the basic anomaly measures, however if you want to interface with a packet analyser, you will need to configure that as it is resource intensive.

If you have `tcpdump` available on your system, it is also possible to get CFEngine to interface to it an measure general packet types travelling on the local network.

```
body monitor control
{
tcpdump => "false";
tcpdumpcommand => "/usr/sbin/tcpdump -t -n -v";
}
```

CFEngine monitoring is normally very lightweight, but occupying the network interface for packet analysis is quite the opposite!

The following example bundle shows we can use the classes set the by monitoring daemon simply to warn about anomalous statistical states. Notice that we define what anomalous means in the policy itself, by setting a class based on the current state in relation to the learnt state, e.g. `rootprocs_high_dev1`, meaning that the current state is between zero and one standard deviation above the expectation value, or average. An 'anomaly' is defined syntactically to be more than two standard deviations above or below average.

Entropy measures can also be used to see how much entropy (variation) there is in the source IP addresses from which the connections arise. This allows us to distinguish between focused traffic from a single source, and traffic arriving from many sources.

```
bundle agent anomalies
{
reports:

rootprocs_high_dev1::

  "RootProc anomaly high 1 dev on $(sys.host) at $(mon.env_time)
   measured value $(mon.value_rootprocs) with
   average $(mon.av_rootprocs) pm $(mon.dev_rootprocs)"

   showstate => { "rootprocs" };

 entropy_www_in_high.www_in_high_anomaly::

   "HIGH ENTROPY Incoming www anomaly high anomaly dev!! on
   $(sys.host) at $(mon.env_time) - measured value $(mon.value_www_in)
   av $(mon.av_www_in) pm $(mon.dev_www_in)"

      showstate => { "incoming.www" };

 entropy_www_in_low.anomaly_hosts.www_in_high_anomaly::

   "LOW ENTROPY Incoming www anomaly high anomaly dev!! on
   $(sys.host) at $(mon.env_time) - measured value $(svalue_www_in)
   av $(av_www_in) pm $(dev_www_in)"

     showstate => { "incoming.www" };

 entropy_tcpsyn_in_low.anomaly_hosts.tcpsyn_in_high_dev2::

   "Anomalous number of new TCP connections on $(sys.host) at
   $(mon.env_time)  - measured value $(mon.value_tcpsyn_in) av
     $(mon.av_tcpsyn_in) pm $(mon.dev_tcpsyn_in)"

     showstate => { "incoming.tcpsyn" };

 entropy_dns_in_low.anomaly_hosts.dns_in_high_anomaly::

   "Anomalous (3dev) incoming DNS packets on $(sys.host)
     at $(mon.env_time) - measured value $(mon.value_dns_in)
      av $(av_dns_in) pm $(mon.dev_dns_in)"

     showstate => { "incoming.dns" };

 anomaly_hosts.icmp_in_high_anomaly.!entropy_icmp_in_high::

   "Anomalous low entropy (3dev) incoming ICMP traffic on $(sys.host)
    at $(mon.env_time) - measured value $(mon.value_icmp_in)
    av $(mon.av_icmp_in) pm $(mon.dev_icmp_in)"

     showstate => { "incoming.icmp" };

}
```

2.3 Threshold monitoring

The values learned by the monitoring daemon are made available to the agent as variables in the system context 'mon', e.g. '$(mon.www_in)'. We can use these values at any time to set policy, either for alerting or fixing the system.

```
vars:

 "probes" slist => { "www", "smtp", "ssh" };

classes:

 "$(probes)_threshold" expression => isgreaterthan("$(mon.$(probes)_in)","50");

reports:

  "Help me $(probes)!" ifvarclass => "$(probes)_threshold";
```

The value of the CFEngine's component integration is to have transparent access to the detailed running state of the system at all times. It is possible to defined quite complex policies, and to respond to incidents based on the specific data discovered.

2.4 Port monitoring

The cf-monitord sets some system variables that allow you to see what ports are in use on a local host. These data are stored locally in '/var/cfengine/state/env_data' and may be seen in variables reports in the Mission Portal.

The following are list variables:

'mon.listening_ports'
: A list of all open ports

'mon.listening_tcp4_ports'
: A list of open TCP ports bound to IPv4.

'mon.listening_tcp6_ports'
: A list of open TCP ports bound to IPv6.

'mon.listening_udp4_ports'
: A list of open UDP ports bound to IPv4.

'mon.listening_udp6_ports'
: A list of open UDP ports bound to IPv6.

The following are scalar array variables:

'mon.tcp6_port_addr[*port*]'
: Variable contains the value of the IPv6 address bound to the port with number given by the array slot.

'mon.tcp4_port_addr[*port*]'
: Variable contains the value of the IPv4 address bound to the port with number given by the array slot.

This excerpt shows list variables and an array of addresses for the bindings to each port, both for IPv4 and IPv6.

```
@listening_ports={'80','5308','631','22','53','1194'}
@listening_tcp6_ports={'631','22','53','80'}
@listening_tcp4_ports={'5308','631','22','53','1194'}

tcp6_port_addr[631]=::1
tcp6_port_addr[22]=::
tcp6_port_addr[53]=::
tcp6_port_addr[80]=::
tcp4_port_addr[5308]=0.0.0.0
tcp4_port_addr[631]=127.0.0.1
tcp4_port_addr[22]=0.0.0.0
tcp4_port_addr[53]=0.0.0.0
tcp4_port_addr[1194]=127.0.0.1
```

This simple policy extract will generate a list of open ports on a given host:

```
reports:
  monitoring::
  "Open tcp4 port on $(mon.listening_tcp4_ports)";
  "Open tcp6 port on $(mon.listening_tcp6_ports)";
```

Sample output:

```
R: Open tcp4 port on 5308
R: Open tcp4 port on 631
R: Open tcp4 port on 22
R: Open tcp4 port on 53
R: Open tcp4 port on 1194
R: Open tcp6 port on 631
R: Open tcp6 port on 22
R: Open tcp6 port on 53
R: Open tcp6 port on 80
```

3 Log scanning

In CFEngine Nova and above, you can extract data from the system in sophisticated ways from files or pipes, using Perl Compatible Regular Expressions to match text. The `cf-monitord` agent is responsible for processing measurement promises.

In this example, we count lines matching a pattern in a file. You might want to scan a log for instances of a particular message and trace this number over time.

3.1 Scanning log files for patterns

You will have to scan the log file for each separate summary you want to keep, so you win a lot of efficiency by lumping together mulitple patterns in a longer regular expressions.

Be careful however about the trade-off. Disk access is certainly the most expensive computing resource, but a smart filesystem might do good caching.

Regular expression processing, on the other hand, is CPU expensive, so if you have very long or complex patterns to match, you will begin to eat up CPU time too.

At the end of the day, you should probably do some tests to find a good balance. One goal of CFEngine is to minimally impact your system performance, but it is possible to write promises that have the opposite effect. Check your work!

```
bundle monitor watch
{
measurements:

   "/home/mark/tmp/file"

         handle => "line_counter",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log("MYLINE.*"),
   history_type => "log",
         action => sample_rate("0");

}

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

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body action sample_rate(x)
{
```

```
ifelapsed => "$(x)";
expireafter => "10";
}
```

3.2 Scanning syslog for FTP statistics

There are many things that you can set CFEngine at monitoring. For example, CFEngine can automtically collect information about the number of socket-level connections made to the ftp server, but you might want more detailed statistics. For example, you might want to track the volume of data sent and received, or the number of failed logins. Here are a collection of monitoring promises for doing just that.

Note that the ftp logs are maintained by syslog, so it is necessary to match only those lines which correspond to the appropriate service. We also assume that the specific messages are sent to '`/var/log/messages`', while your configuration may specify otherwise. Likewise, your operating systems's version of ftp may issue messages with a slightly different format than ours

```
bundle monitor watch_ftp
{
vars:
   "dir" slist => { "get", "put" };

measurements:

   "/var/log/messages"

        handle => "ftp_bytes_${dir}",
   stream_type => "file",
     data_type => "int",
   match_value => extract_log(".*ftpd\[.*", ".*${dir} .* = (\d+) bytes.*"),
  history_type => "log",
        action => sample_rate("0");

   "/var/log/messages"

        handle => "ftp_failed_login",
   stream_type => "file",
     data_type => "counter",
   match_value => scan_log(".*ftpd\[.*", ".*FTP LOGIN FAILED.*"),
  history_type => "log",
        action => sample_rate("0");

   "/var/log/messages"

        handle => "ftp_failed_anonymous_login",
   stream_type => "file",
```

```
     data_type => "counter",
   match_value => scan_log(".*ftpd\[.*", ".*ANONYMOUS FTP LOGIN REFUSED.*"),
  history_type => "log",
        action => sample_rate("0");

}

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

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body match_value extract_log(line, extract)
{
select_line_matching => "$(line)";
extraction_regex => "$(extract)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

3.3 Scanning DNS logs for query statistics

Another thing you might want to do is monitor the types of queries that your DNS server is being given. One possible reason for this is to test for unusual behavior. For example, suddenly seeing a surge in 'MX' requests might indicate that your system is being targeted by spammers (or that one of your users is sending spam). If you are thinking of converting to IPv6, you might want to compare the number of 'A' requests to 'AAAA' and 'A6' requests to see how effective your IPv6 implementation is.

Because DNS logs are directly maintained by 'bind' or 'named' (and do not go through syslog), the parsing can be simpler. However, you *do* need to configure DNS to log query requests to the appropriate log file. In our case, we use '/var/log/named/queries'.

```
bundle monitor watch_dns
{
vars:
    "query_type" slist => { "A", "AAAA", "A6", "CNAME", "MX", "NS",
                            "PTR", "SOA", "TXT", "SRV", "ANY" };
measurements:
```

```
    "/var/log/named/queries"
        handle => "DNS_$(query_type)_counter",
        stream_type => "file",
        data_type => "counter",
        match_value => scan_log(".* IN $(query_type).*"),
        history_type => "log",
        action => sample_rate("0");
}

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

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

3.4 Scanning syslog for email statistics

Email is another syslog-based facility that you may want to use CFEngine to monitor. There are a number of volumetric data that are of interest. For example, the number of messages sent and received, the number of messages that have been deferred (a large number might indicate networking problems or spam bounces), and the number of spam messages that have been detected and removed by the assorted spam filters.

The samples below assume that there is a separate logfile for email (called '/var/log/maillog') and that a few of the standard sendmail rulesets have been enabled (see 'http://www.sendmail.org/~ca/email/relayingdenied.html' for details). As with any syslog-generated file, you need to check for the appropriate service, and in this case we are lumping local messages (sent through 'sm-mta') and remote messages (sent through 'sendmail') into a single count. Your mileage may of course vary.

If you use one or more sendmail "milters", each of these will also output their own syslog messages, and you may choose to track the volume of rejections on a per-filter basis.

```
bundle monitor watch_email
{
vars:
    "sendmail" string => ".*(sendmail|sm-mta)\[.*";

    "action" slist => { "Sent", "Deferred" };
```

```
measurements:

   "/var/log/maillog"

         handle => "spam_rejected",
    stream_type => "file",
      data_type => "counter",
                   # This matches 3 kinds of rulesets: check_mail,
                   # check_rcpt, and check_relay
    match_value => scan_log("$(sendmail)ruleset=check_(mail|rcpt|relay).*"),
   history_type => "log",
         action => sample_rate("0");

   "/var/log/maillog"

         handle => canonify("mail_$(action)",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log("$(sendmail)stat=$(action) .*"),
   history_type => "log",
         action => sample_rate("0");

}

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

body match_value scan_log(line)
{
select_line_matching => "$(line)";
track_growing_file => "true";
}

body action sample_rate(x)
{
ifelapsed => "$(x)";
expireafter => "10";
}
```

3.5 Scanning syslog for email milter failures

Milters are relatively new in sendmail, and some have problems. You can also use monitoring to detect certain types of failure modes. For example, if a milter is running (that is, there is a process present) but it does not respond correctly, sendmail will log an entry like this in syslog (where 'xyzzy' is the name of the milter in question):

```
Milter (xyzzy): to error state
```

A small number of these messages is no big deal, since sometimes the milter has temporary problems or simply encounters an email message that it finds confounding. But a larger value of these messages usually indicates that the milter is in a broken state, and should be restarted.

You can use '`cf-monitord`' to check for the number of these kinds of messages, and use the soft classes that it creates to change how '`cf-agent`' operates. For example, here we will restart any milter which is showing a high number of failure-mode messages:

```
bundle monitor watch_milter
{
vars:
   "milter" slist => { "dcc", "bogom", "greylist" };

measurements:

   "/var/log/maillog"

         handle => "${milter}_errors",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*Milter (${milter}): to error state"),
   history_type => "log",
         action => sample_rate("0");
}

bundle agent fix_milter
{
vars:
    "m[dcc]" string      => "/var/dcc/libexec/start-dccm";
    "m[bogom]" string    => "/usr/local/etc/rc.d/milter-bogom.sh restart";
    "m[greylist]" string => "/usr/local/etc/rc.d/milter-greylist restart";

commands:
    "$(m[$(watch_milter.milter)])"
        ifvarclass => "$(watch_milter.milter)_high";
}
```

3.6 Scanning syslog for breakin attempts

A lot of script-kiddies will probe your site for vulnerabilities, using dictionaries of account/password combinations, looking for unguarded accounts or accouts with default passwords. Most of these scans are harmless, because a well-maintained site will not use the default passwords that these hackers seek to exploit.

However, knowing that you are being scanned is a good thing, and CFEngine can help you find that out. Because '`sshd`' logs it's message through '`syslog`', we again need to filter lines based on the service name. On our system, authorization messages are routed to '`/var/log/auth.log`', and we would monitor it like this:

```
bundle monitor watch_breakin_attempts
{
measurements:
    "/var/log/auth.log"
         # This is likely what you'll see when a script kiddie probes
         # your system

         handle => "ssh_username_probe",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*Invalid user.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # As scary as this looks, it may just be because someone's DNS
         # records are misconfigured - but you should double check!

         handle => "ssh_reverse_map_problem",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*POSSIBLE BREAK-IN ATTEMPT!.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # Someone is trying to log in to an account that is locked
         # out in the sshd config file

         handle => "ssh_denygroups",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*group is listed in DenyGroups.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # This is more a configuration error in /etc/passwd than a
         # breakin attempt...

         handle => "ssh_no_shell",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*because shell \S+ does not exist.*"),
   history_type => "log",
         action => sample_rate("0");
```

```
    "/var/log/auth.log"
         # These errors usually indicate a problem authenticating to your
         # IMAP or POP3 server

         handle => "ssh_pam_error",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*error: PAM: authentication error.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
         # These errors usually indicate that you haven't rebuilt your
         # database after changing /etc/login.conf - maybe you should
         # include a rule to do this command: cap_mkdb /etc/login.conf

         handle => "ssh_pam_error",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*login_getclass: unknown class.*"),
   history_type => "log",
         action => sample_rate("0");
}
```

See the CFEngine Nova documentation for more possibilities of measurement promises.

4 Intrusion detection

Intrusion detection is a highly specialized area. CFEngine is not designed specifically to be an intrusion detection or intrusion prevention system, but it has many features that can be used as part of an integrated stratey against intrusions.

What is an intrusion or an attempted intrusion? This can be difficult to define. If someone tries to login as root once? If someone tries to login at root fifty times? Is port scanning a sign, or perhaps a SATAN or ISS scan? Someone trying a known security hole? There is no certain way to identify the intentions behind activity we observe on a system.

The aim of an intrusion detection system is to detect events that can be plausibly connected to incidents or break-ins, hopefully while they are still in progress so that something can be done about them. One way of doing fault diagnosis is to compare a system to a working specification continuously. This is essentially what CFEngine does with systems.

CFEngine®