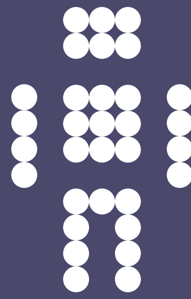
CFEngine



Role Based Access Control and CFEngine

A CFEngine Special Topics Handbook

CFEngine

Role Based Access Control (RBAC) is a well known paradigm for granting privileged access to remote command systems. The paradigm of RBAC does not apply directly to CFEngine because CFEngine is an autonomous system that does not grant change privileges to external users. Role-based read-privileges can be granted through the CFEngine Nova Mission portal.

This document is aimed at security experts who are trying to understand transference of privilege in a CFEngine managed system.

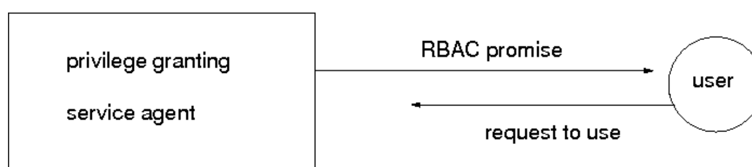
Copyright © 2011 CFEngine AS

What is Role Based Access Control?

Role Based Access Control (RBAC) describes a set of promises made by a host to grant privileged access to the system. In this regard, RBAC is no different from any other form of access control, however, it is normally used to grant the privilege to execute certain commands that make changes to the system – thus it involves *write* or *change* privilege.

The term *role*-based is used because users are often classified into managerial roles that are each assigned different levels of privilege with regard to the kind of tasks they need to perform.

Role Based Access Control is used when remote users request access to a privileged service from some kind of service-agent running on a host. For example, the password on the Unix root account is a simple RBAC system where access is granted to execute any command with unlimited privilege, to any user who knows the root password.



RBAC is about remote action privilege

The risks of RBAC

Granting privilege to execute commands has obvious risks. The implementation of restricted access is usually handled in one of a number of different ways. The term RBAC does not explain, in itself, which of these models will be used.

Two common alternatives may be distinguished:

Arbitrary commands may be executed by privileged individuals (trusted user model).
: This is a high risk privilege granting system, where arbitrary change privileges are granted to users.

Pre-defined privileged operations may be made accessible to certain individuals/roles (Clark-Wilson model).
: This is a lower risk system, where users are only allowed privilege while executing very specific pre-defined activities (e.g. the right to initiate a backup).

CFEngine's approach to privilege

CFEngine handles privileged access somewhat differently. To see why, it is important to understand what CFEngine is not:

- CFEngine is not a service agent that grants remote change access to systems.
- CFEngine is not a remote-control for system operations[1].

CFEngine is intended for autonomous hands-free operation, and thus the issue of remote access almost never arises directly. In CFEngine it is expressly forbidden for a part of CFEngine

[1] Many provisioning and management systems are indeed effectively remote execution agents and thus RBAC is more relevant to them.

to receive commands from external parties. In a limited sense, CFEngine can be configured to listen for requests for classes that label the context for extraordinary, predefined policies; these can then be activated by certain users (`roles` promises), providing a version of the second form of RBAC above. We shall return to this below.

By design it is not possible to send instructions to CFEngine that have not been pre-approved by the host administrator and promised as policy. Ultimately the local host administrator can veto any proposals for change in any configuration system (e.g. by unplugging the network).

In CFEngine this is made a central tenet of the management model.

From a security perspective, the elimination of remote command access presents a huge simplification to security, without loss of functionality. The risk of executing privileged commands on the system is exchanged for a right to submit policy changes. Thus the access control becomes a matter of *who is allowed to approve policy for dissemination* to the system.

The chain of privilege in CFEngine

Even though CFEngine is not in the business of granting privilege for command execution, there are security implications to using CFEngine and thus we should examine the chain of influence from user to host to understand the implications.

In normal usage, users work as follows:

- A user edits a CFEngine input file.
- The input file determines a promise proposal, or template for policy.
- Someone publishes the policy for dissemination to interested parties.
- Interested parties (hosts running CFEngine) may choose to download these new propsals from trusted sources. These sources are defined in the existing policy which may always be vetoed by a local administrator.
- CFEngine will execute the new policies with the maximum privilege granted to it. This privilege can be altered:
    - By virtue of the privilege with the CFEngine itself runs.
    - By the privilege accorded to a promise as a matter of its own policy.

When planning for the security of system changes, one should assume that any promise written in a CFEngine policy will be enforced with maximum privilege.

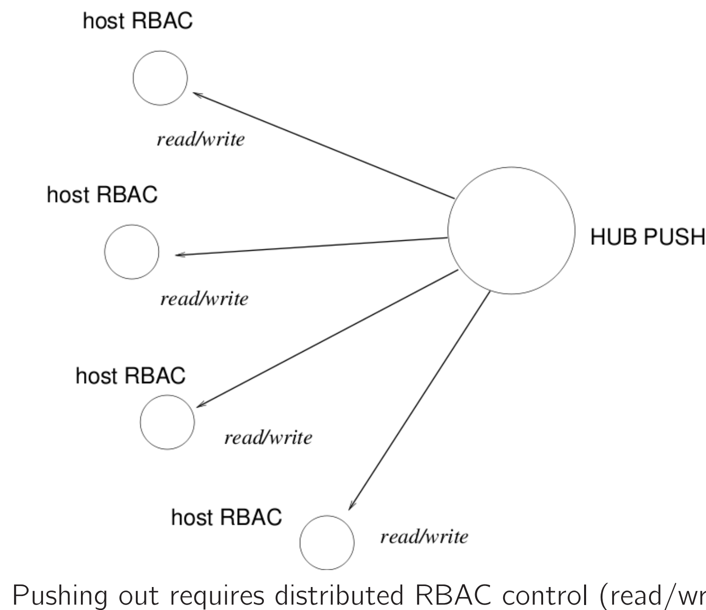
The role of centralized push and pull in RBAC

Centralization is a strategy of collecting resources into a single location. A central resource often becomes authoritative for a collection of hosts. Centralization has positive and negative aspects

- As a single point of direction, it simplifies the coordination of multiple agents (like a conductor in an orchestra).
- It can be a single point of failure, and a bottleneck for operations. Becauses centralization concentrates effort, brute force must be used by a hub when scaling to many hosts around a centralized strategy.

In terms of privilege, the implications of centralization are significantly different for *push* and *pull* based systems (see the figures below).

Let us first consider the general problem, without reference to CFEngine.



Pushing out requires distributed RBAC control (read/write).

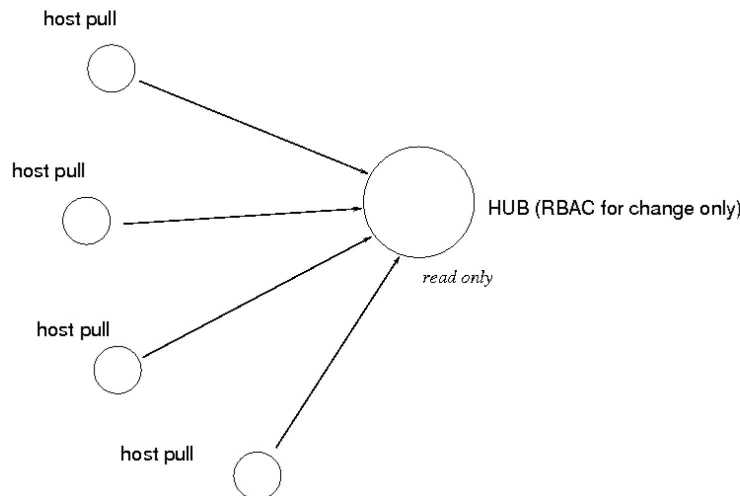
A *push* is defined to be either

- The involuntary transmission of data to hosts from a hub, or
- The remote execution of commands from the hub to the hosts.

We see easily from the figures below that configuration of adequate access control requires access control configuration to be implemented on every managed host. If the hosts require different levels of access in different zones, for instance, this requires a distributed configuration control of the RBAC system itself across the affected hosts. There is thus a burden to setting up RBAC.

One configures a system for a *push* system just as one configures a system against attack from outside. In configuration terms, push is indistinguishable from an attack.

Pull-based management is fundamentally different. In a pull model, hosts download public information (their policy) from a trusted source.



Pulling updates requires only centralized RBAC control for change,
but not for the update itself (read only).

There is no need for access control on the hosts anymore, since they are only reading information voluntarily. They may simply reject all attempts to send them data, in favour of their voluntary decision to download updates.

Moving from push to pull-based configuration simplifies the number of independent points of configuration from N to 1, and the location of access control information is simplified from N separate models to a single model at the hub. The hub can decide which hosts will have access to which policy proposals, so there is no loss of privacy: the security model's definition is fully centralized (single point of definition for consistency).

- Push-based approaches have centralized execution, but distributed RBAC configuration of the management setting. Multiple, inconsistent pushes from different hubs can even lead to distributed inconsistency that cannot be detected from any single location.
- Pull-based approaches have distributed execution, but a single point of security configuration. Inconsistent pulls are impossible, as there is a single point of definition.

At CFEngine, we strongly believe that pull-based systems are superior for most purposes, because pull's distributed operation reduces the risk of the bottleneck, while the centralized definition of access rights reduces the risk of error.

In all further sections, we assume CFEngine's pull-based model.

No need for any centralization in CFEngine

Before continuing, it is important to emphasize that CFEngine has no technological *need* for centralization. The decision to centralize management is a policy decision. Every host can, if desired, be configured as an independent device, with its own policy, making no contact with any external host. CFEngine is thus ideal for embedded systems and environments with partial connectivity, such as ships and submarines. Nevertheless, centralized management is often chosen for its simple coordination of decision making. What is important to realize is that centralized decision-making is a convenient fiction for managers – no remote party can truly decide the state of a host.

The owner of a machine always has the privilege to make changes to it. Push-based models of management that pretend to control hosts absolutely are simply misleading, as they exist by the good grace of end systems.

In the remainder of this Special Topics Guide, we shall assume the common model of centralized management, because that is the context in which RBAC is relevant.

The risk from centralized trusted hosts

Centralization has implications for risk[2]. Gaining malicious control of a trusted source could have a significant impact on all the hosts that subscribe to updates from it.

The risk, in this case, is precisely the same as that for a push-based system that executes certain commands. However, the task of defending a single trusted host is (at least psychologically) simpler than that of defending all the hosts in the network[3].

The risk of propagating a bad change (i.e. an unfortunate mistake) is also no different between push and pull. A bad decision is simply a bad decision. The antidote to human errors is to conduct policy reviews, i.e. use more pairs of eyes, or 'dual-key' solutions.

Centralize the writing of policy, within a local region to obtain straightforward consistency. Don't overcentralize, or you will oversimplfy. One size rarely fits all (see the *Special Topics Guide on Federation and Organizational Complexity*). RBAC then becomes an issue of: who should have the right to edit and publish changes to policy?

The Policy Dispatch Point

The burden of security is now localized entirely at the Policy Dispatch Point. It becomes the responsibility of this 'role' (policy dispatcher) to ensure that the desired state is in fact the one that is promised. This happens in two practical steps:

- Editing of an SVN repository of working proposals (access to change respository).
- Merging of changes into the actual published policy (bowtie process).

Where the highest levels of paranoia are justified, no host should receive automatic updates of policy without explicit human inspection and policy review. This is equivalent to allowing no RBAC privileges.

The right to edit and publish policy

Let's recap' for a moment. The CFEngine agent runs with maximum system privilege (root/Administrator), and makes its decisions based on a set of promise proposals that come from some trusted source, e.g. the owner of the machine, or some central policy decision point. Once a set of proposals has been published, we simply call these 'the policy'. The agent on each host reads these proposals and picks out those that are relevant to the current context ('here and now') for each host. The agent then tries to keep these promises, by

[2] A single point of definition could also be a single point of failure. In CFEngine, a central policy hub is not a point of failure, because each agent caches all the resources it needs to maintain systems according to its current model. At worst, the loss of a hub would mean a delay to updates.

[3] User who are adept at automated configuration might disagree, as automation makes it easy to harden all hosts equally well. Network policies such as firewalls, etc, are however, simpler to manage for a single host.

making any necessary changes to the system. For most common usages of CFEngine, the effect is that anything that is in the published policy is executed with up to maximum privilege.

This means the following:

Any user who can edit the actual source policy has control over a host.
: The policy should not be writable by any unauthorized user, in any location where it will be picked up as part of the policy-appoved process for updating policy[4].

Any user who can cause a new version of the policy to be published for immediate use has privileged access.
: RBAC now means limiting access to the files that define policy.

If policy is automatically checked out of a repository, commit access to the respository can give privileged access.
: There should be a process of approval for changes made to policy. This should be a human process, because ultimately a human must be responsible for publishing a policy. In this situation, RBAC now consists of granting access to make commits to the repository.

The conclusion of this section is that only a small number of highly trusted individuals should be able to alter policy themselves.

Distributed coordination. RBAC is a poor tool for delegating tasks alone, because if multiple individuals with access rights are not coordinated in their promises, the result will merely be a conflict.

The bowtie process

Promise theory allows us to model the collaborative security implications of this (see the figure of the bow-tie structure). A simple method of delegating is the following.

1. Delegate responsibility for different issues to admin teams 1,2,3, etc.
2. Make each of these teams responsible for version control of their own configuration rules.
3. Make an intermediate agent responsible for collating and vetting the rules, checking for irregularities and conflicts. This agent must promise to disallow rules by one team that are the responsibility of another team. The agent could be a layer of software, but a cheaper and more manageable solution is the make this another group of one or more humans.
4. Make the resulting collated configuration version controlled. Publish approved promises for all hosts to download from a trusted source.

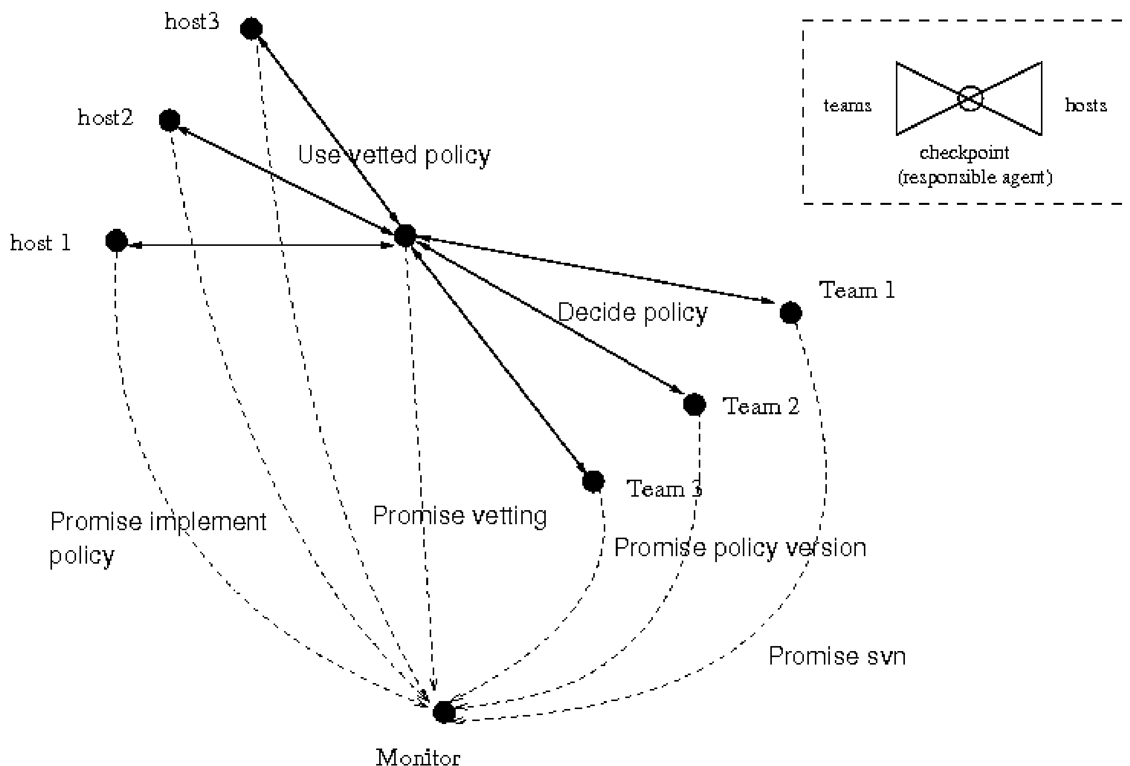
A review procedure for policy-promises is a good solution if you want to delegate responsibility for different parts of a policy to different sources. Human judgement as the 'arbiter' is irreplaceable, but tools can be added to make conflicts easier to detect.

Promise theory underlines that, if a host or computing device accepts policy from any source, then it is alone and entirely responsible for this decision. The ultimate responsibility for the published version policy is the vetting agent. This creates a shallow hierarchy, but there

[4] Note that the decision to collect policy updates from somewhere is itself a policy decision in CFEngine, so users should always think carefully about these decisions.

is no reason why this formal body could not be comprised of representatives from the multiple teams.

The figure below shows how a number of policy authoring teams can work together safely and securely to write the policy for a number of hosts, by vetting through a checkpoint, in a classic ‘bow-tie’ formation.



Granting the right to switch on special pre-defined policies

CFEngine offers one technological convenience that is relevant to RBAC. In the Clark-Wilson security model, non-privileged users can be granted limited privilege to execute predefined commands that are locked down to specific actions. The Unix `ps` and `passwd` commands are examples of this, for example.

Most users do not need to touch CFEngine at all, because policy is checked very regularly and promises are enforced with 5 minute intervals. In other words, for most users, just waiting will fix anny problem. In some cases, there are extraordinary promises or tasks that one does not want implemented without human oversight. In that instance, one places the relevant promises in a context that is not normally active. Users can then activate those sleeping promises by defining the context class manually.

```
bundle agent mybundle
{
files:

  extraordindary::
```

```
    # ... promises ...
}
```

Privileged users who have access to the system do not need RBAC to do this as they already have all credentials they need, and can achieve the same thing by running the agent with a defined class, e.g.

```
host# cf-agent -D extraordinary
```

However, it is also possible to grant access to these parts of a CFEngine policy that are normally switched off by using `cf-serverd` to mediate privilege to execute the agent with this class active. For example, setting:

```
bundle server access_rules()
{
roles:

  # Allow mark

  "extraordinary"  authorize => { "mark", "sally" };
}
```

and running:

```
host# cf-runagent -H special_host -D extraordinary
```

would achieve the same effect without granting any rights to change the policy.

In this example CFEngine promises to grant permission to users '`mark`' and '`sally`' to remotely activate classes matching the regular expression '`extraordinary`' when using the `cf-runagent` to activate CFEngine. In this way one can implement a form of Role Based Access Control (RBAC) for unprivileged users, provided users do not have privileged access on the host directly. User identity is based on trusted CFEngine keys created by the user and exchanged with the server.

RBAC-filtered read-access in CFEngine Nova

CFEngine Nova 2.2.0 introduces Role Based Access Control (RBAC) for all reports and promises shown in the Mission Portal. This does not cover access control for making policy changes, but displaying reports.

RBAC can be globally switched on or off in the Mission Portal settings.

Authentication

User-authentication is carried out when users log in to the Mission Portal. This is done by requiring a user name and password, which is checked against the following possible sources.

- Internally defined in the Mission Portal
- LDAP

- Acitive Directory

The selection between these options are available in the Mission Portal settings.

Authorization

The information a user is authorized to see is determined from his role memberships. A user may be member of an arbitrary number of roles, each which may grant and deny access to certain information.

The effective permissions of a user is the cumulative set of permission granted or denied by his roles, and is used to filter the information displayed in the following standard way.

- Create a union of the granted access for the roles.
- Override with the rules that deny access for the roles.
- If left unspecified, access is denied.

Entities filtered

RBAC is supported on the *host* and *promise bundle* level, each applying to different parts of the Mission Portal. Both these entities are atomic with respect to RBAC — either a user can see everything they contain, or nothing of it.

Access to a host is required to see any information about it, e.g. all its reports (Engineering->Reports), host page, and compliance category. If a user is not allowed access to a host, the Mission Portal would look the same as if the host was not bootstrapped to that hub.

Information about the running policy is also available in the Mission Portal, either through the Promise Finder at the Engineering page, or by clicking a promise handle from one of the reports. The searchable promises in the Promise Finder and information pages about promises and bundles are filtered in the same manner as the hosts, but defined based on promise bundles instead. The Policy Editor is not covered by RBAC — access to the policy source repository allows the user to see the whole policy. Some version control systems can be configured to only allow users to access sub-directories of the policy, which may help in this case.

Note that the host and promise filtering is independent — no attempt is made to try to infer which promises a role should have access to based on the hosts it has access to or vice versa.

Defining roles

From the above discussion, we see that a role is defined as reporting access to a set of hosts and promise bundles from the Mission Portal and REST API. This does not give any rights with respect to changing the content or execution of the policy. It should not be confused with the `roles` promise-type that can be used by `cf-runagent` and `cf-serverd`.

In order to scale, both entities are defined as a set of *regular expressions* to allow and deny.

Access to hosts is defined by regular expressions on *classes*, not the hostname, ip, or any other name. This is done to ensure maximum scalability. Classes can be arbitrarily defined in the CFEngine policy language, so this incurs no loss of flexibility, but ensures distributed computation.

In contrast to users, a role definition and membership can only be obtained from the internal Mission Portal database. This means that any roles must be defined through the Mission Portal

web interface, and can not be obtained from e.g. LDAP at this time. The rationale is that querying complex LDAP structures for role membership is too inefficient and error-prone. This may change in future releases, if requested. Note that the *possible members* of a role can be obtained from other sources, as described in '`Authentication`' above. However, assigning possible members to roles must be done through the Mission Portal user-interface.

A sample definition of the role '`lob_a`' is shown below.

Only members of the '`admin`' role has the ability to manipulate roles and their memberships.

After defining the role itself, the next step is to make the designated users members of the role, using the Mission Portal.

Limitations

- Notes added in the Mission Portal are not filtered: they can be seen by all users (including notes added to any host page).
- The Knowledge Map is only available for members of the '`admin`' role when RBAC is switched on.
- Running `cf-report` from the command-line on the hub will bypass all RBAC checks.