

Overall Design of Policy Related Components

From Idmwiki

Contents

- 1 Policy Related Concepts
 - 1.1 Application
 - 1.2 Types of Policies
 - 1.2.1 Configuration Policies
 - 1.2.2 Actions
 - 1.2.3 Roles
 - 1.2.4 Comparison of the Types of Policies
- 2 Policy Management
 - 2.1 Defining Policy
 - 2.1.1 Editing Configuration Policies
 - 2.2 Associating Policies with Hosts
 - 2.3 Preview of the Policy
 - 2.4 Policy Validation
 - 2.5 Policy Life-cycle Management
- 3 The Policy Engine: Under the Hood
 - 3.1 Policy Format
 - 3.1.1 Metadata
 - 3.1.2 Main Body
 - 3.1.2.1 Configuration Policy
 - 3.1.2.2 Roles
 - 3.1.2.2.1 General Overview
 - 3.1.2.2.2 Roles for Policy Kit-Enabled Applications
 - 3.1.2.3 Actions
 - 3.2 Relax NG
 - 3.2.1 Policy Schema
 - 3.2.2 Documenting Schema
 - 3.2.3 Schema and UI
 - 3.2.4 Schematron
 - 3.3 Special Tags in Policy Schema
 - 3.4 Storing Policies
 - 3.5 Storing Schemata
- 4 IPA Client and Policy Delivery
 - 4.1 Local XML Processing Files
 - 4.2 File System Structure
 - 4.3 Merge Rules
 - 4.3.1 Merging XML
 - 4.3.2 Merging Local Files
 - 4.3.3 Backup and Restore Logic for local Files
 - 4.4 Reloading Configurations
 - 4.5 Handling Roles
 - 4.6 Handling Actions
 - 4.7 Handling Policy Deletes
 - 4.8 Policy Lookup Logic in DS
 - 4.8.1 Lookup Roles
 - 4.8.2 Lookup Configurations
 - 4.8.3 Lookup Actions
 - 4.9 XML-RPC Interface
 - 4.10 XML-RPC Connection

- 5 Developing and Testing Policies
- 6 Policies for the IPA Client
- 7 DS Schema for Policy Related Objects
- 8 Adding Support for New applications
 - 8.1 Adding New Application Roles
 - 8.2 Adding New Configuration Policies

Policy Related Concepts

One of the requirements of IPA v2 is to provide centralized policy management. "Policy", however, is a very loaded term and can mean different things in different contexts. During our analyses of the use cases we came to the conclusion that there are different types of policies that need different kinds of treatment. We started by sorting policies into different buckets based on their characteristics.

- The first type of policy that we identified is host-based access control policies (HBAC). These are the policies that define which users can access which machines. The main characteristic of a HBAC policy is that it should be very dynamic and reflect reality as much as possible. Any delayed processing and caching should be minimized to avoid granting a user access to a host when his permissions to access that host have just been revoked. It was decided that for dynamic policies like HBAC, when a change to the user's group membership can significantly affect the scope of what he can do or access, it should be stored in the DS and clients should do LDAP searches every time with very minimal caching if the system is online. In the offline case, the cached HBAC policies will be used (if the IPA client policy allows it). The HBAC design is covered in detail on the Concepts and Objects page. The schema objects and low-level implementation details can be found on the Representing Objects in DS (Part 1 – Base objects) page.
- The second type of policy is user policies. These are policies that are related to the user and affect the user environment regardless which machine he is logged on to. In the Windows world this is called a "Roaming Profile" - a collection of settings that follow the user irrespective of which machine he uses to log in. This kind of policy is not part of v2. It is deferred to a later version.
- The third and the major type of policy is the machine policy. This is the type of policy that affects the state of a machine or a group of machines. The idea behind this type of policy is that the policy is defined on the server, delivered to the client and then translated into a specific configuration of the client software. This configuration can affect what users can do, but permissions for a specific user can be different on different machines. It is assumed that these policies are not frequently changed, and thus clients can pull and apply these policies periodically – once an hour, for example.

The rest of this page is dedicated to machine policies and drills down into the details of how these policies are defined, stored, maintained, delivered, and applied.

Application

One of the important concepts that we introduced during our analysis and evaluation of machine policies is the concept of an application. The idea is that the policy usually makes sense in the context of an application, or, to put it simply, "policy translates into a specific application configuration following a centrally defined rule". We see the mission of IPA's policy engine as centralized management of the policies (read security configurations) of the different security-related applications like SUDO, SELinux, iptables, and others. Potentially IPA can be used for broader configuration management but this is not currently a goal for IPA. Although IPA is being designed with broad flexibility and extensibility in mind, it is yet to be seen and proven if IPA would be the right instrument for a broader configuration management task.

The concept of an application allows for the sorting of policies and dealing with configuration on per-application basis. Consequently, the SUDO policy translates into the configuration of the SUDOERS file, the apache policy into a collection of apache-related files and so on.

From the policy management perspective, the IPA administrator would use the web UI (or CLI), select the application, define the policy (configuration) for this application and then associate this policy with a set of hosts that should have this configuration. The policy engine and IPA client will do the rest – deliver the

policy to the client and translate it into the application configuration.

Sometimes, however, the policy should be viewed more broadly than on a per-application level; that is, across several applications. To address this we plan two different solutions.

- The first solution is considered for IPA v2. If time permits we will implement so-called "policy profiles". A policy profile is a collection (group) of policies defined for different applications that are in some way related to each other and should be delivered to the same set of hosts. From the policy management perspective, this would mean that the IPA administrator would have to:
 - define individual application policies
 - aggregate application policies into a group of policies (policy profile)
 - apply (assign) this group of policies to a set of hosts

This is a convenience feature to better organize policies and provide a more logical management experience. Without this feature the administrator would have to apply each individual (application-related) policy to a set of hosts.

- The second solution is to provide a so-called "meta policy". The idea behind "meta policies" is that their sole purpose would be to become a portal to changing related configuration settings for different applications at the same time. There are several ideas how this can be accomplished but this feature is not planned for IPA v2 and thus is not discussed in more detail here.

Types of Policies

As mentioned above, we are talking about centrally-managed policies that define the configuration (configuration state) of an application on a client machine. While evaluating different use cases, we realized that there are other situations besides delivering configurations to applications that the policy engine needs to handle. One of these examples is SELinux. SELinux policies are compiled binary blobs. Delivering and applying such a policy means downloading a policy file from the IPA server and running a script or command to apply it. This use case led to an attempt to sort out the different types of policies and responsibilities the policy engine should be responsible for. The following subsection dives into the three different responsibilities.

Configuration Policies

Configuration policies are the policies that, as was mentioned above, are defined and managed on the IPA server itself. They are delivered to the client and applied using specific client components that we will discuss in more detail below. The following list summarizes the characteristics of the "configuration policies":

- defined on the server itself (not an externally defined blob or file that needs to be distributed)
- defined in the context of an application
- can have multiple instances of the policy that target different machines
- can be more than one policy in the scope of an application. These policies can target overlapping sets of hosts. For example, consider a default policy that applies to a large set of machines, and also a specific policy that applies to a subset of machines. These two policies need to be merged on those machines to create a resulting combined policy (configuration).
- can be associated with overlapping sets of hosts and thus require merging. To be able to merge policies predictably the precedence rules must be defined for these types of policies.
- more or less free-structured content. The content of the policy is dictated by the application and there are no requirements for its structure or format.

Actions

"Actions" is a different type of policy. It is actually more an operation than a policy. There are always two different views on the system configuration:

- One view is state oriented - "here is the configuration, configure the application like this on the client so that the final state of the configuration matches the specified policy".
- Another view is action- or operation-oriented - "run this script to apply the SELinux policy"

The best practice is to view the configuration as a state but in reality both mechanisms – “change state to be like this” and “perform action” are needed. The “actions” in IPA give the administrators the capability to centrally control the execution of different operations and apply policies that are not managed directly by IPA. In this case IPA just provides the delivery mechanism and framework to do something with delivered data or script. We envision the action as a combination of download and run operations. The download portion of the action is about what file to download (if any), what it should be called, its location on the target system, ownership and permission; what condition should be met to actually download the file; what script to run (if any) under which user context and how frequently.

The following list summarizes the characteristics of the actions:

- Actions deal with operations not with state
- Actions can perform download and execute operations. Any action can have either part or both.
- Actions have a single format but not all the data can be defined if the action is related to just download or just execution.
- Actions can be applied to different sets of the machines
- More than one action can be targeted to a machine but there is no need to merge. Actions are independent of each other and should be executable in any order. If an action is dependent on another action then these actions should be viewed as one action and combined.
- Although actions can sometimes be logically associated with applications there is no sense to bind them together. The main value for application binding for the configuration policies is that their priorities can be defined in the scope of the application. Actions do not need priorities and also one action can affect several applications at the same time.

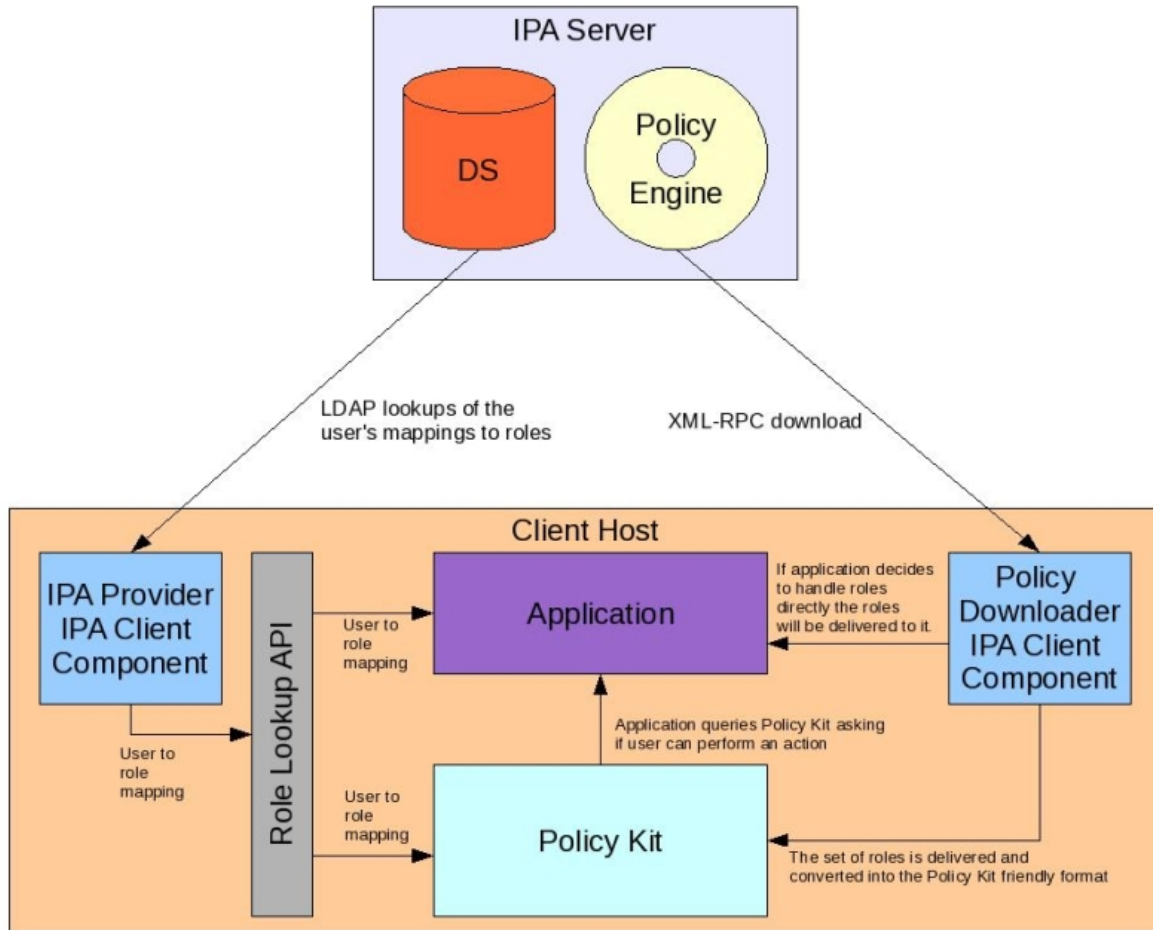
Roles

"Roles" is the third category of policies that the policy engine will deal with. Many applications need to deal with the question of whether or not a user can perform a specific action on a machine inside this or that application. Currently this kind of access control is implemented on a per-application basis. IPA's role mechanism would allow the application to take advantage of the centralized role management defined in IPA instead of maintaining or building its own role-based access control solution. The application usually knows the user that tries to perform an action and what an action means. The application is usually interested in just a yes or no answer about whether the user can perform the action or not. The desktop team developed a library called "Policy kit" that applications (currently desktop applications mostly) can use and ask whether a user can perform an action. The mapping between users and actions in Policy kit is currently done based on direct user association with actions on a local machine. The data is stored on each individual machine in the files that contain direct mappings of users to actions. With IPA v2 a new level of abstraction will be created. The operations will be aggregated into roles and users, and groups of users will be mapped to roles. The Policy kit will then receive role information from IPA via the policy engine. On the other side users will be mapped to application roles in IPA. The Policy kit will integrate with the IPA client and will use IPA client interfaces that would return to the caller the set of roles the user is mapped to in the context of an application. The following list illustrates what would happen if an application decides to take advantage of the IPA backed up Policy kit.

- In the policy engine of IPA an administrator will define roles (usually once per application). Roles should be viewed as named (tagged) sets of permissions/operations one can perform if he or she logs in to an application. Canned role definitions will probably be provided for different applications as the product matures.
- The administrator will define the mapping of users to different roles. The mapping follows the notion that users can assume different roles within the same kind of application running on different machines. So to define a role the administrator will associate a user or group of users and a host or group of hosts to a role (role tag) in the context of an application.
- The IPA policy engine will deliver the policy (definition of roles) to IPA client machines. The IPA client will translate this information into the format that is more suitable for Policy Kit to use.
- The application integrated with the Policy kit will request an access control decision from the Policy Kit.
- In turn the Policy Kit will ask the IPA client about what roles the user is mapped to on the current machine in the context of an application. The IPA client will provide an interface to get user roles. The details about this interface can be found on the [Overall_IPA_Client_Design](#) page.
- By matching user -> role -> operation to each other, Policy Kit will determine if the user can perform the operation or not.

Applications are not required to use Policy Kit. Applications can take advantage of IPA's role infrastructure directly processing role data and mapping users to roles on its own.

The following diagram gives an overview of roles and related concepts in IPA v2.



The following list summarizes the characteristics of roles:

- Roles are named sets of operations/permissions one can perform in the context of an application.
- Roles make sense in the context of an application
- There is only one set of roles per application
- The role set should be delivered to all the machines that are going to host an application these roles are assigned to.
- Since there is only one definition of a role per application there is no need to merge or prioritize role definitions.
- Roles can be inclusive or exclusive. An inclusive role means that the user can have several roles in the context of the application and the permissions defined by these roles should be merged. The logic for merging role meanings (sets of permissions) in this case is application-specific and left to the Policy Kit or application itself to implement. If the application supports exclusive roles, then only one role can be assigned to the user at a time.
- Role definitions should have a specific format so that UI engine could pull the possible role names from it. This would create a list of all possible roles that users can be mapped to. It would be convenient for the UI that would define the mapping of users to roles.
- The mappings of users to roles will be stored in the DS and will be looked in the DS with caching for a very short period of time. This will be done to dynamically adjust user capabilities when a user is moved from one group to another.

Comparison of the Types of Policies

As noted above, the policy engine will be in charge of configuration policies, actions and roles. All these parts, though significantly different, have the following things in common:

- the same mechanism will be used to store and manage information on the server
- delivery of the information will be performed via the same channel – the “policy downloader” client component

Policy Management

In this section we will talk a little bit about the anticipated user experience with the management of the policies.

Defining Policy

Depending on the task, the administrator will:

- Define a new or modify an existing configuration policy for an application
- Define a new action
- Define or change application roles

Editing Configuration Policies

To edit configuration policies the administrator will navigate to the appropriate place in the UI (menu item) that will allow him to define configuration policies. There he will select the application for which he wants to define a policy. Then he will see the list of already defined policies within the context of the application. There the administrator can change the order of the policies in the list to change policy priority, associate a policy to a group of hosts, select a policy for editing, copy or remove a policy or start editing a new policy.

The policy data for each application will be different. To accomplish such flexibility, the policy engine should be extensible to allow dropping in different policy templates. The UI engine will use these templates to render the UI and allow the administrator to set values for the policy.

Potentially there can be several different configuration policies that have not only different values for the same configuration parameters, but also define values for different configuration parameters. For example, there can be a default policy that defines a value for parameter X while there is another policy applicable to the same application that defines a value for parameter Y. All the configuration policies for an application are prioritized in one list. If two different policies are assigned to the same host they will be merged. It is anticipated that usually there will be one set of the configuration parameters expressed by the same template (or different versions of such). In this case the merge logic will assume that the policies can be merged and the data is complementary. On rare occasions when the application would require more than one set of configuration data, i.e. different templates, it should be split into two, different, unrelated applications from a management perspective. In future we might allow more than one configuration template per application.

The term *template* used in this section is a way to describe the fact that the policies will be flexible and there will be a way to define the contents of the policy and tell the UI how to collect policy data from a user. In reality there will be no templates but rather policy structure definitions, called *schema*. We will talk about this in more detail in later sections of this page.

Associating Policies with Hosts

Once a policy is defined, the administrator will have the option of associating the policy with a set of hosts. This set can be expressed as a mixture of hosts and host groups. The administrator will select hosts and host groups from a list. As mentioned above, there is a chance that two different policies will be associated with the same subset of hosts. In this case the policies will be merged based on the merge rules defined for a policy.

Preview of the Policy

It is very important for an administrator to understand how the data he entered into the policy form would actually translate into the configuration file or files on the client. In IPA v2 we plan two kinds of preview. One preview will just create a configuration file out of the selected policy and would allow the administrator to check if everything is Ok with the policy he is working on. Another preview mechanism will show a preview of the policy in the form of how it will be delivered and saved in the configuration files on the target host. This preview will request the administrator to enter the target host and would perform merges of the policies if necessary. The preview will not show what would be the result of the merge of the central policy with the local files on the host if local merges are allowed. This feature is deferred to future releases.

In addition to the preview of the policy itself, the preview will actually show how the merge was done. It is not clear how exactly it will be done but the vision is to have a log that would contain something like this:

```
-----
Evaluated SUDO policies for host ipa-client@test.lab.com
3 policies are destined for host ipa-client@test.lab.com
  Default Corporate Policy - precedence 4
  Department Policy - precedence 16
  Lab Policy - precedence 39
SUDO application does allow merging policies
Default Corporate Policy was selected as a base for merge
  Department Policy was merged on top
  Lab Policy was merged on top
-----
```

With such a log the administrator would be aware of the logic used and how things are going to be merged.

Policy Validation

It is important to realize that the policy UI (and CLI) would not have any specific business logic that will validate that one instance of the policy does not contradict another instance of the policy. There will be syntactical validation and validation of the consistency of data within the policy itself, but not across multiple policies. Multiple policies will be merged based on the merge rule defined for the policy and the priorities of these policies. The customer has two options:

- develop and test policies in the test IPA domain
- use the preview mechanism to determine what policy will be downloaded to the client

It is important to underline that in v2 the preview mechanism does not support the preview of the merge of local files with the centrally defined policy. In future this might be addressed, most likely by special command line utilities that would allow this kind of merge preview if run from the host the policy targets.

Policy Life-cycle Management

In the future we plan to provide full policy life-cycle management. In IPA v2 we will provide part of this management:

- the system will support two policy states - enabled and disabled. This means that disabled policies are not sent to clients, only the enabled ones. This is mostly useful for the case when the whole policy should be disregarded, without actually deleting it.
- the policy engine will have two-step commit functionality. One would be able to write a policy but not apply it yet. This is useful when the policy needs to be modified and prepared but applied later, during outage windows, while the old version of the policy is active. The administrator will have the capability to edit the policy and save it and then apply it later. In future we might consider doing scheduled commits. The UI that of the policy list will be built in such a way that it would be easy to specify if you want to use an uncommitted policy in preview or not. This would allow the administrator to see if the policy is correct in the context of other, already applied policies before pushing it out.
- the policy will have a special field that would contain comments about the changes made to the policy.
- there will be a one-depth history for the policy. This functionality would be useful if a policy were applied and something stopped working, and there was a need to roll the policy back to the last known good state. In the future we will extend the history depth to be configurable.
- there are no plans to support an approval mechanism for the submission of policies in IPA v2. This

functionality will be considered for later releases.

The Policy Engine: Under the Hood

Here we discuss the mechanisms that will be used to define, manage, store, and deliver the policies. The policy will be represented by a blob of XML data. The following section delves into the details of how the XML policy will look.

Policy Format

XML was selected for its flexibility and extensibility. Each policy XML file will have a structure that will be common for all the policies of the same type (config policies, roles, actions).

Metadata

Firstly, each policy, regardless of type, will contain a metadata section at the top of the file. The metadata will hold the information about this policy, including, but not limited to:

- the version of the policy
- the name of the policy
- who created this instance of the policy

The following example shows a subset of the metadata:

```
<metadata>
  <name>
    simple sudoers example, allowing mount/umount of a CD-ROM
  </name>
  <author>sbose@redhat.com</author>
  <version>0.7071</version>
  <RNGfile>sudoers.rng</RNGfile>
  <XSLTfile>sudoers.xslt</XSLTfile>
  <app>SUDO</app>          <- means that this policy is for SUDO
  <mergeXML>yes<mergeXML>  <- means that in case of collision the XML files need to be merged
                           (alternative means the policy with higher precedence wins)
  <local>no</local>        <- do not merge with local files - applicable to the configuration section only
</metadata>
```

The example shows the kind of metadata that will be stored in the policy file. It is currently unclear what the final set of required and optional metadata elements will be. It will be published separately as prototyping of different policies proceeds.

Main Body

There are three different types of XML policy file – one for each type of policy we identified earlier.

Configuration Policy

For the configuration policy the main body will contain the configuration data for an application. This configuration data is different for different applications so there is no predefined structure of this information. To read more about how the structure of the XML file is defined read the Relax NG section later on this page.

The following is an example of the configuration policy for sudoers:


```

<sudoers>
  <subject>
    <name>abc</name>
    <type>netgroup</type>
  </subject>
  <command>
    <path>/sbin/umount /CDROM</path>
    <tag>NOPASSWD</tag>
    <runas>root</runas>
  </command>
  <option>
    <authenticate>on</authenticate>
  </option>
  <command>
    <path>/sbin/mount -o nosuid,nodev /dev/cd0a /CDROM</path>
  </command>
</sudoers>
<sudoers>
  <subject>
    <name>def</name>
    <type>posixGroup</type>
  </subject>
  <option>
    <authenticate>off</authenticate>
  </option>
</sudoers>
<sudoers>
  <subject>
    <name>EWLKFKJKFwe</name>
    <type>ALL</type>
  </subject>
  <command>
    <path>/sbin/shutdown -r now</path>
  </command>
  <option>
    <lecture>always</lecture>
  </option>
</sudoers>

```

This snippet is specific to the SUDOERS file and SUDO's configuration policy. For a different configuration policy the structure will be completely different.

Roles

General Overview

The structure of the XML files that define roles will be a little bit more formal. The reason for this is to make it easier for the IPA UI engine to parse the XML blob and extract the names of the roles so that they can be used in other places in the UI, especially when someone tries to map users to the roles.

The following example shows how roles can be organized. It is an example and subject to change as we do the prototyping of the role-defining XML files:

```

<roles>
  <role name=Common>
    <...>
  </role>
  <role name=Uncommon>
    <...>
  </role>
  <role name=Default>
    <...>
  </role>
</roles>

```

< Roles section

<- Role specific data

<- Role specific data

<- Role specific data

The suggested structure of the XML roles file is partially fixed. The data under the `role` element can be anything the application needs, but the other data and especially the elements that define the role name should follow a predefined structure so that they can be easily extracted from the file.

Roles for Policy Kit-Enabled Applications

Each application will have its own definition of roles and what they mean. However, the applications that leverage Policy Kit will have more in common. Initially we considered treating Policy Kit as an application and making it contain one role definition for all other applications that will leverage Policy Kit. Such an

approach is not flexible and thus was rejected. Each application that leverages Policy Kit will have a similar template for the roles so that it is easier to develop new role definitions for new Policy Kit-enabled applications. All Policy Kit-enabled applications define roles based on the application actions (do not confuse this with IPA actions described above). Each application that integrates with Policy Kit defines its own set of actions. On the local host the actions are part of the application and installed when the application is installed. On the server side it is required to know these "actions" too so that they can be used in the role definition policy. There are several options as to how these actions can become known to the server side. We have evaluated the following options:

- Store application actions for Policy Kit in the DS - this approach involves a lot of overhead. Further, looking at the picture, storing the Policy Kit actions in DS does not fit the high-level architectural approach. It would also require the policy processing engine to be much smarter and allow dynamic lookup of the data in DS. We plan to implement dynamic lookups but only for host object attribute substitution. If we go the LDAP route we would have to develop a much more complex feature.
- Store the application action list hard-coded in the schema - this approach requires knowing all the actions in advance. It does not scale well since modifications to an application will require the IPA administrator to update the schema for the application and redistribute the new schema to all the clients.
- Store the possible actions in a special section of the role definition policy itself - this approach calls for a more generalized schema that can hold the list of actions that are defined within the scope of the application. It would be possible to populate this list manually in the UI or by loading it into IPA from the file that came with the application. This approach seems most promising. It means that the role file schema will also contain a special section after the metadata and before the role definitions that will list all possible actions that can be used later in role definitions in the roles section of the role definition policy.

Actions

The main purpose of actions is to perform an operation on the client system. Sometimes the action requires some data to be delivered to the client before the action should be executed. This data can be the policy to apply, a configuration file to copy or a script to execute. After analysis and comparison of different use cases, we came to the conclusion that the "action" should consist of a file section and an execution section. Only one of these sections is required. This means that the action can specify only that a file should be downloaded, that a script should be run, or both.

The file section describes what to do when the data is delivered to the client system. It specifies:

- the data itself (which is read from the file when the policy is defined by the administrator and embedded as an element into the XML blob)
- the full path of the file that needs to be created on the client system
- the ownership of the file
- the permissions on the file
- the SELinux label on the file
- the ACL for the file
- condition - a script or command that needs to be run on the client. If that script returns zero or a certain output then the file should be downloaded. The condition allows for differentiating client systems based on their properties, for example the version of OS or version of the application. It has not yet been decided whether there will be one condition targeting both file download and action, or if there will be two separate conditions, one controlling the file download and another controlling the execution of the command. It is proposed that if the file download does not happen because the condition was not met, then the command should not run. With this assumption in mind there should be only one condition that applies to both the file and the command.
- cleanup flag - should the file be removed after the action was successfully run

The "command" section of the file will contain the command itself and the "schedule" element. The "schedule" element will define how frequently and under what conditions the command should be executed.

The action XML blobs will have the same pre-defined structure for all the actions that can be defined in the system. The structure of the action XML blob might look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<ipa xmlns="http://freeipa.org/xml/rng/ipaaction/1.0">
  <metadata>
    <name>simple ipaaction example with an URL</name>
    <author>sbose@redhat.com</author>
    <version>0.7071</version>
    <RNGfile>ipaaction.rng</RNGfile>
    <XSLTfile>ipaaction.xslt</XSLTfile>
    <app>ipaaction</app>
  </metadata>

  <ipaaction>
    <condition>
      <command>test -e /etc/redhat-release</command>
    </condition>
    <file>
      <url>http://my.server.org/something.txt</url>
      <path>/tmp/something.txt</path>
      <owner>nobody</owner>
      <group>nogroup</group>
      <access>0444</access>
      <selinux_context>unconfined_u:object_r:user_home_t:s0</selinux_context>
      <acl>user:dummy:rw-</acl>
      <acl>user:admin:rw-</acl>
      <cleanup>no</cleanup>
    </file>
    <run>
      <command>/bin/rm /tmp/something.txt</command>
      <user>admin</user>
    </run>
  </ipaaction>
</ipa>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<ipa xmlns="http://freeipa.org/xml/rng/ipaaction/1.0">
  <metadata>
    <name>simple ipaaction example with embedded data</name>
    <author>sbose@redhat.com</author>
    <version>0.7071</version>
    <RNGfile>ipaaction.rng</RNGfile>
    <XSLTfile>ipaaction.xslt</XSLTfile>
    <app>ipaaction</app>
  </metadata>

  <ipaaction>
    <file>
      <data>VGhpncyBpcyBhIHRlc3QK</data>
      <path>/tmp/something_other.txt</path>
      <owner>nobody</owner>
      <group>nogroup</group>
      <access>0444</access>
    </file>
  </ipaaction>
</ipa>

```

The action XML blob structure is not yet finalized, but this gives a good example of what constitutes an action. It is not clear how far we will go with the implementation of the scheduler in the action. As an option there might be a series of configuration policies that would translate into the crontab configurations that would be equivalent in functionality to the actions but would require a bit more customer involvement.

The file to download in the file section can be specified as embedded data and stored as binary data in the policy itself. In this case the action will contain the <data> element. Alternatively the file to fetch can be expressed as a url. In this case the action processing part of the policy downloader will fetch this file using the specified URL.

Relax NG

In the previous section we looked at the different formats of the XML policy blobs depending upon the type of policy this blob represents. In this section we will discuss how to define and enforce the structure. Historically there have been several ways of defining the structure of an XML document. During the early evaluation of the alternatives we identified Relax NG as the best option for definition of the structure of the XML blobs. Relax NG is an XML-based language used to define the structure of an XML file. Relax NG was selected for its flexibility and ease of use.

Policy Schema

The following Relax NG schema gives an example of how the configuration policies will be defined using Relax NG schema.

Complete_sudo_Policy_Example

Documenting Schema

The Relax NG schema language supports namespaces. One of the namespaces is used for annotation of the schema itself.

Schema and UI

The Relax NG schema can not only contain the description of the XML file structure but also the description of the UI that needs to be used to interact with the user and collect from him the values for the policy fields. A special namespace is used to annotate which UI elements should be used to represent different data elements that need to be collected from the user.

Exact elements and tags are currently being designed and will be added later as we move forward with prototyping of the policy-rendering UI.

Schematron

In addition to RNG, we will use Schematron. XML Schema, Relax NG, and others are prescriptive in that they stipulate what elements may appear, in what order and with what content. Schematron is rules-based and allows expression of constraints that the other schema languages cannot, such as "when element <foo> has value <bar>, element <baz> must not exist". Schematron by itself is almost never a good choice, but it is common to use it as an adjunct to the other schema languages. Relax NG has direct support for embedding Schematron annotations.

Special Tags in Policy Schema

In addition to the UI and documentation tags, the schema can in future contain tags for audit so that special events are logged not only when the policy is saved or applied but when specific values in the policy change. It is unclear whether we will have time to implement this feature in IPA v2.

Another valuable feature we will consider implementing in IPA v2 is references from the policy to the host object attribute. The idea behind this feature is that it might be useful to have a policy be a template and have it customized based on the attributes of the host object the policy is delivered to. The current vision of this feature is that the schema for a policy will contain a special tag that will have the name of the host object. Then the client will do the substitution of this tag with the real value from the host entry. It is unclear if we will have time to implement this feature in IPA v2.

Storing Policies

Now that we know and understand what format the policies are stored in, and how we will enforce the structure of the policy, we can talk about the storage mechanisms used for storing policies. As mentioned above, the policy itself is an XML blob. We have looked at two options for how the XML blobs will be stored in the system. The first option was to use the file system (as Microsoft does); the second was to put the XML blobs into the DS itself. In both cases the DS should have "helper" or so-called "link" entries that help find the right policy. While comparing the two options, we kept in mind that:

- there will be a DS entry with a link to the policy
- we need to think about replication
- we need to think about scalability of the solution and performance

The file-based approach has several significant issues but also some advantages.

Pros:

- it can store big chunks of data
- it is scalable

Cons:

- it is necessary to replicate the changes separately from the changes to the policies expressed as XML blobs stored as files.
- potential for de-synchronization between DS and file system
- a lot of work related to replication
- the available file replication packages are not robust enough for our needs
- file-based access control (different from DS)

The DS approach has the following characteristics:

Pros:

- consistent multi-master replication out-of-the-box
- much less work
- the same DS-based access control rules as for other objects in the system

Cons:

- scalability concerns if policies become really big

We have chosen the second option for v2. We will implement it in such a way that we can later switch to using file system if we face the scalability issues.

We plan to compress the policies before saving them in the DS entry. XML has a very good compression ratio. Because policy management is not a frequent task, and because it is expected that new versions of policies will be developed in the test environment, the impact of the compression and decompression operations would be minimal.

Storing Schemata

The schemata for the policies will be stored in XML files on the file system and will not be replicated. This means that when a new schema for a new application is added, the schema definition files should be copied to all replicas manually. This should be viewed as a system upgrade and as such it is acceptable to require installation of the schema files on different replicas manually and not to provide automatic replication of the schema definitions for policies. In the future we will consider storing the schemata in the DS too but this is currently outside of the scope of IPA v2.

A more detailed description of how to add support for policies for new applications can be found later on this page.

IPA Client and Policy Delivery

The API client will use two components: the IPA Provider and the Policy Downloader to perform policy downloads. It is a pull model. The Policy Downloader will periodically ask the IPA data provider for the list of new policies that it needs to download. It is expected that the IPA Provider will be an LDAP “guru”, and would be in charge of doing different kinds of sophisticated DS lookups. The actual logic of those lookups will be defined later on this page. As a result of the lookup, the data provider will return the list of policies the Policy Downloader should download. The Policy Downloader will connect to the IPA server using the XML-RPC mechanism. The reason we will use XML-RPC mechanism is to abstract the storage mechanism. We anticipate that we might have to change the storage mechanism in the future from DS-based to some other method if we see scalability problems.

The Policy Downloader will request the policies based on the returned list. The policies will be requested via XML-RPC call one by one. Each policy will be delivered compressed. The client code will decompress the policy, perform any special tag substitution and store the resulting XML blob in the file in the system. The file system directory hierarchy is covered later on this page.

In the current proposal the Policy Downloader and the data provider are two separate components that both connect to the IPA server. It makes sense to logically separate the duties of these two components because one is responsible for doing the DS lookups and the other for downloading and processing the policies. Physically we might consider combining them into a single process. The idea is that it would be easier to update the client and integrate it with other data and policy providing back-ends like Samba or a 3rd-party central server. Regardless of whether or not these two components are combined there is pretty much the same amount of work if keeping them separate or together under one process umbrella. The current plan is to implement them as two separate processes in IPA v2. Later they can be refactored for better extensibility.

We also evaluated the approach of making the Policy Downloader XML-RPC calls rely on server-side processing rather than asking the data provider to do the searches. This puts a lot of burden on the server, especially in deployments with many clients. It is hard to assess which approach is better without prototyping. The current plan is to do the lookups using the data provider and do the processing of the policy lists on the client. In the future we might re-evaluate this approach if we face performance and scalability problems.

Configuration policies that are destined for a single application will be merged based on the merge rules. The resulting policy will also be stored in a special place in the file system. Then the Policy Downloader will launch the XML converter, passing in the XML filename as an argument. The XML converter is the utility that would translate the XML policy into the format the application expects. The current plan is to have a converter utility that will use an XSLT template to translate the contents of the XML policy to some other format more native to the application. The XSLT is a type of XML-based program that defines how to process the XML file and convert it to something else. Every application will have its own XSLT template to use with the converter. What template to use will be defined in the policy itself in the metadata section as shown earlier on this page. The name of the XSLT template in the metadata of the policy will be populated automatically based on the schema. It is possible for the application to have more than one XSLT that needs to be applied. In this case the metadata will have multiple XSLT templates specified and the Policy Downloader will launch the conversion utility with each of the specified XSLTs.

So far we envision the following XSLT templates that we would provide:

- XSLT template for SUDO files
- XSLT template for IPA client configuration
- XSLT template for SELinux roles

etc.

For the text files after the conversion is done there can be yet another step – merging the configuration file locally with the local configuration file. The rule for merging the files is based on the flag stored in the policy's metadata. If the flag is omitted the policy should assume that IPA is the authoritative source of the configuration information for this application and the local configuration file should be overwritten. If the flag indicates that the files should be merged then the Policy Downloader will launch the local file merger utility. There will be no generic local file merge utility. The complexity of the utility would depend on the configuration files it needs to merge. For simple merges a simple shell script can be used. For more complex merges we plan to use the Augeas library. To read more about Augeas project see <http://augeas.net/>. Specific merge logic will be implemented as we prototype local file merges. The ability to merge with local files is the property of an individual policy. In the UI an administrator will be able to specify if the policy should be merged with local files or not. This creates a situation where there might be two different policy instances destined for a host: one can be merged and the other cannot. A special attribute of an application object will define whether the result of merge of such instances should or should not be merged with local files.

Local XML Processing Files

For the Policy Downloader to be able to process the XML file and for the converter to translate the policy into something else the schema definition and XSLT file shall be installed on the client system for every supported application. For the applications that will be supported out-of-the-box, the Relax NG and XSLT files would be a part of the distribution. For the applications for which the IPA integration will be enabled later the files would have to be distributed. For more details of how this can be done, refer to the Adding Support for New Applications section at the bottom of this page.

File System Structure

The downloaded XML files after extraction and file substitution will be stored under the directory structure in the following directory:

```
/var/cache/ipa/policy/xml
```

Under this there will be three subdirectories:

- config – for configuration files
- roles – for roles
- actions – for actions

Under the “config” directory there will be subdirectories with the application names. Under the application directory there will be files with names that correspond to the policy's unique identifier. For more details about policy's unique identifiers, refer to the DS Schema and Policy Related Objects section later on this page. There will also be one policy called `final.xml`. This policy is created as a result of merging the policies. If there is only one policy destined for the machine and there is no need to merge, the `final.xml` file will be a link to that policy file.

Under the “roles” directory there will also be subdirectories with the application names. Each application directory will contain one file (if any) that will describe the roles. Because there is never more than one file with a role definition per application, there is no need to merge role definitions. However, an application might expect the roles to be translated into some other format so there will be a converter run when the role definition is downloaded.

The “actions” directory will not have any subdirectories. All the actions will be stored in the “actions” directory itself. The name of the action policy file will be based on the policy's unique identifier.

The file's creation or modification time stamp will be used to determine whether or not a new version of the policy needs to be downloaded. Alternatively (if we have time), we can implement the sequence-based approach to determining if the policy needs to be refreshed. For details about policy download logic, refer to the Policy Lookup Logic in the DS section later on this page.

Note: Renaming applications would not be supported and any attempt to do it manually might cause problems on clients. This should be an acceptable trade-off because application names should be treated as an internal part of the system.

Merge Rules

There are two kinds of merges that the system might perform:

- merging several application configuration XML policy files
- merging the centralized policy with the local policy

Merging XML

In IPA v2 we will support only two XML file merge options. The first one is not really a merge. It just states that the policy with the higher precedence wins. Such an approach is useful when for any application there might be several instances of the policies with similar contents but assigned to different groups of hosts that can potentially overlap.

The other option is an additive merge. This would mean that the policy with the lowest precedence will be taken as a base and all the other policies that need to be merged with it will be laid over it, overwriting common data attributes. As a result, a combined policy will be created. The merge algorithm will be the same on the client and in “preview” on the server so that the administrator can actually see the results of the merge operation as he develops the policy on the server.

How the policies should be merged for an application is a property of the application and thus will be stored in the application entry. For more details about DS schema for application and other related objects refer to the DS schema topic later on this page.

In the future, other merge methods might be supported by the system. It is also planned to create a pluggable framework for handling the merges. This feature is outside the scope of the IPA v2 project.

We also evaluated the option of allowing individual policy instances to be marked as ones that can be merged and ones that can't be merged. We think that this functionality can be accomplished by creating two different types of applications: one with XML policy instances that can be merged between each other and one that can't. Allowing a mixture of the two would complicate the UI and the merge logic and confuse the user. Because it is possible to add this complexity incrementally in the future, this idea has been deferred to later versions.

Merging Local Files

The policy can be the authoritative source of information. In this case the configuration files generated based on the policy will just replace the local configuration files. In another case there might be a need to respect the local files and merge the local file and the central file.

There are two different pieces of information that control the merge logic with local files:

- whether the merge is needed. This information should be specific to the policy instance, because it is possible that one policy instance targeting one subset of machines should be merged but another targeting another subset of machines should not. A flag in the metadata of the policy instance will indicate whether the merge is needed. By default, if there is no flag present in the XML file then the client will assume that the central policy takes precedence. This flag should be editable in the UI if this option makes sense for the application. In cases where one policy will be translatable into several local files, the rules about merges should be defined on a per-file basis inside the corresponding XSLTs.
- what should be merged with what. It is possible that the configuration created based on a policy should be merged with a local file or vice versa; that is, the local file shall be merged to it. This kind of information is part of the policy as a whole and can't be different on different machines since it depends on the structure and nature of the policy itself. This will be specified in the code of the XSLT template. In case the policy shall be translated into multiple configuration files, the XSLT logic for each file will have the right code to do the right thing for that kind of file.

We plan to use Augeas for the merging of files. We would create lenses (sets of regular expressions used by Augeas) which Augeas would use to parse the file and present the configuration in a form of a tree. The task of merging the configuration files becomes the task of merging two trees together.

Backup and Restore Logic for local Files

Whenever the local file is merged or replaced, it is important to keep an original (or last known good copy) of the configuration file that was on the system. It is needed in case the policy is changed so the merge has to be performed again or in case the policy is completely rolled back. Then the client shall restore the original configuration file.

The logic described here shows how we will handle backing up local files. If several files are affected by the policy, this logic will be applied to each of the files.

- if there is no backup file (thatis, this is first time the policy touches the file):
 - copy the configuration file into `/var/lib/ipa/policy/backup/<application>` with appended extension `.orig`
 - copy the configuration file into `/var/lib/ipa/policy/backup/<application>` with appended extension `.backup`
 - create a new configuration file in place of the old one by performing a merge or just replacing the file depending upon the values of the flags that were discussed in the previous section
 - run a one-way hash (sha256 for example) of the resulting generated file. Convert the hash into a printable hexadecimal string. Append this string to the name of the backup file. This is done to record the state of the file and to be able to make the right decision if the file is manually altered by the administrator.
- if there is a backup file and the hash string of the backup file is the same as the hash string of the current configuration file, then there was no out-of-band modification of the file. This means that our backup file is current, so if we need to perform the merge the backup file will be used as a base for the merge. If we do not need to perform a merge then the current configuration file should just be replaced by the one generated based on the policy.
- if there is a backup file and the hash string of the backup file does not match the hash of the current configuration file, this means that the configuration file was manually modified by someone who had the

authority to change it. We then assume that whoever did it approved the changes so this new file should be treated as a latest correct client configuration. Actions:

- copy the configuration file into `/var/lib/ipa/policy/backup/<application>` with extension `.backup`, overwriting the previous backup file
- create a new configuration file in place of the old one by performing a merge or just replacing the file depending upon the values of the flags
- run a one-way hash (sha256 for example) of the resulting generated file. Convert the hash into a printable hexadecimal string. Append this string to the name of the backup file.
- if the policy is completely removed then restore the original file in place of the configuration file and clean the contents of the `/var/lib/ipa/policy/backup/<application>` directory

One of the alternative ideas about the backup file location is to append the full path of the configuration file into the backup path. So instead of backing up the configuration file for some “test application” called `/etc/testapp/testapp.conf` to `/etc/ipa/policy/backup/test application/testapp.conf.backup.HASHSTRING` as suggested above, the Policy Downloader would back it up into `/var/lib/ipa/policy/backup/test application/etc/testapp/testapp.conf.backup.HASHSTRING`. This approach solves the problem of potential naming collision if there is more than one file that needs to be updated per application. The final decision on which method to use will be made during the implementation phase.

Reloading Configurations

When the policy is delivered, merged, converted into files, backed up, and merged again, it is still not active. Some applications would reread the configuration periodically but some require a restart of the application to apply a new configuration. The XSLT template for the policy can optionally have a command that will be executed to apply the policy. This could be sending a signal to an application, restarting it or doing something else that is prescribed by the application that will cause it to reload its configuration.

Handling Roles

The roles will be handled by running the converter using the XSLT template specified in the role file's metadata. Also, there can be an “apply” command (like in the config case above) that will most likely just copy the resulting converted role file into an application-specific area. It would be the responsibility of the application to deal with the interpretation of the roles.

Some of the applications might choose to take advantage of the Policy Kit in terms of access control checking. The IPA client will have native integration with the Policy Kit. This integration will be two-fold. Firstly, IPA will be able to download role definitions for different applications that will be handled by Policy Kit and store them in the LDB storage (local LDAP style storage). Secondly, IPA will provide the plug-in into the Policy Kit back end. This back end will be able to use the downloaded data put into LDB to make authorization decisions. In case of Policy Kit integration, the role file for an application will be translated into the LDAP ldif file and loaded into the LDB database using LDB tools. Running this tool will be specified in the XSLT in the “apply command” section.

Handling Actions

The XML schema for actions is fixed so that processing of all actions is performed by the same engine (logic). As configuration policies, actions will be prioritized, creating a well-defined execution order. This would allow running several operations in a sequence.

Only actions destined for a specified host will be delivered to that host, but the list will be ordered and actions will be executed following that order.

Each individual action (after caching it on the client system in `/var/cache/ipa/policy/xml/actions`) will be processed following this logic:

- if there is a file specified in the action
 - the condition script will be run
 - if the condition is met then the file will be saved in the file system with the provided file permissions and ownership
- if there is an execution portion of the action

- if the command is scheduled to run once, then run it. Next time it will be run only if the action changes.
- if we need to run the action periodically it might be translated into the cron job crontab or handled by the Policy Downloader itself. Because that functionality can also be accomplished by creating crontab configurations via configuration policies, the periodic aspect of the actions might be deferred to later versions. One of the options on this route is to create an IPA component that would actually run different actions on a scheduled basis. This application will be periodically invoked by cron to check if there is an action to perform. From this perspective it can be viewed as a different application and have its own configuration policy in IPA.

Handling Policy Deletes

When a policy is deleted or rolled back on the server, the clients should restore its original state. This is the case for when there are no more configuration policies destined for a machine. In this case the Policy Downloader (with the help of LDAP lookups performed via the data provider) will determine that there are no more policies destined for the current machine in the scope of some application. It will then:

Type of Policy	Actions to perform
Config policy	* Restore original configuration files from backup * Delete all cached XML files from under /var/cache/ipa/policy/xml/<application>
Roles	No rollback action is planned for the roles in v2. In future the roles XML schema can be extended to handle a rollback action. The reason for not providing the rollback is that the application might not behave properly if the roles become completely unavailable.
Actions	If an action is removed on the server it would just be removed from the /var/cache/ipa/policy/xml/actions directory on the client. In future when (and if) we support crontabs we will remove the associated crontabs.

Policy Lookup Logic in DS

Downloading policies to the client will be done in two steps. First, based on a request from the Policy Downloader, the IPA data provider will determine the unique IDs of the policies that need to be downloaded. Then, the Policy Downloader will download each of these policies using an XML-RPC connection.

It would be beneficial to deal with actions, configurations and roles separately. There might even be a different period defined for each type of policy. For example, because roles rarely change, polling for roles more frequently than once every 2-3 hours does not make much sense. The configuration policies might be more time sensitive, so checking for configuration updates might be done once an hour. The checks for new or updated actions might be done once an hour too, or perhaps even more frequently. The general idea is that they can be handled separately and independently. When the machine is booted, it will probably request roles, configurations, and then actions in that order. This means that actions are run last, and use the latest role and configuration information. This order is subject to change, however. This kind of configuration preference is a good subject for inclusion in the IPA client policy that IPA v2 will provide out-of-the-box.

Lookup Roles

The Policy Downloader will first determine that it is time to check for roles. It will inspect the contents of the /var/cache/ipa/policy/xml/roles directory and will create a list of the applications and the role definition files it knows about. Something like this:

```
{
  {application 1, guid 1, version A}
  {application 2, guid 2, version B}
  {application ..., guid ..., version ... }
  {application X, guid X, version Z}
}
```

The IPA Policy Downloader will then request the list of roles destined for the machine from the IPA provider.

The IPA data provider will perform a series of lookups to determine which role files are destined for the current machine.

- it is assumed that the data provider will download and save the host entry that corresponds to the current host. If the cached record has expired, then

the IPA data provider will update it.

- the IPA data provider will then look at the member attribute of the host entry and create an LDAP search request to return the list of policy link objects. The search request will be very similar to the one described in the Host-Based Access Control Design page.
- the link object will contain a pointer to the policy object itself. This level of abstraction is needed if we decide to store policies on the file system or in some other data store. For more details about the schema and DS objects that would represent policies, refer to the schema section later on this page.
- the list of link entries will be returned to the Policy Downloader.

The Policy Downloader will then compare its list of role definition policies with the list of policies that need to be downloaded. It will determine:

- the role definitions that need to be removed. In this case, the Policy Downloader will invoke the cleanup logic, assuming it is defined in the role definition file it is planning to remove. As mentioned above, this functionality might be deferred to a later version.
- the role definition policies that need to be updated. For each of these the Policy Downloader will:
 - issue a request to get the policy via XML-RPC and then get the policy
 - decompress it
 - save it in the `/var/cache/ipa/policy/xml/roles/<application>` directory, overwriting the previous version
 - execute the converter, if any
 - execute the "apply" command, if any
- new policies that need to be downloaded. For these:
 - create a new storage location the under the roles directory
 - <all other actions from "update" use case above>

The logic to compare the two lists will look like the following (the logic is written in some abstract language that is convenient for expressing algorithms; it is not basic or 4GL though it has some splendid similarities):

```

// Prepare the two lists for comparison
// Sort these two lists by application name – assume it is done as we construct both lists.

NeedNextExitingPolicy = TRUE
NeedNextIncomingPolicy = TRUE

WHILE TRUE
  IF NeedNextExitingPolicy = TRUE
    GET NEXT ExistingPolicy
    IF ExistingPolicy NOT AVAILABLE
      // This is the end of the existing policy list
      // So all the rest incoming policies are new
      WHILE AVAILABLE
        GET NEXT IncomingPolicy
        ProcessNewPolicy(IncomingPolicy)
      END WHILE
      // We are done so
      BREAK
    END IF
    NeedNextExitingPolicy = FALSE
  END IF
  IF NeedNextIncomingPolicy = TRUE
    GET NEXT IncomingPolicy
    IF IncomingPolicy NOT AVAILABLE
      // This is the end of the incoming policy list
      // So all the rest existing policies should be removed
      WHILE AVAILABLE
        GET NEXT ExistingPolicy
        DeleteOldPolicy(ExistingPolicy)
      END WHILE
      // We are done so
      BREAK
    END IF
    NeedNextIncomingPolicy = FALSE
  END IF
  // We have both
  IF ExistingPolicy < IncomingPolicy
    // Existing policy should be removed
    DeleteOldPolicy(ExistingPolicy)
    NeedNextExitingPolicy = TRUE
  ELSE IF ExistingPolicy > IncomingPolicy
    ProcessNewPolicy(IncomingPolicy)
    NeedNextIncomingPolicy = TRUE
  ELSE // They are equal
    IF ExistingPolicy.Version < IncomingPolicy.Version
      // We got a new version of the policy
      UpdatePolicy(ExistingPolicy, IncomingPolicy)
    ELSE
      // If it is same version do nothing
    END IF
    NeedNextExitingPolicy = TRUE
    NeedNextIncomingPolicy = TRUE
  END IF
END WHILE

```

This proven algorithm allows comparing two lists in one pass, avoiding $M \times N$ fetches and comparisons. Instead, it uses just $N+M$ fetches and $\text{MAX}(N,M)$ comparisons, thus providing better performance. I encourage us to use this login anywhere we need to compare two lists that can be presorted by the same criteria at the moment of construction. Even if presorting is required as a separate step it is still usually less operations than $M \times N$.

Lookup Configurations

Configurations will be looked up in much the same way as role definition policies above. The only difference is that there will be two lists to deal with: a list of the applications, and a list of the policies inside the applications.

The following example shows what the Existing and Incoming policy lists would look like:

```

{
  {application A, {{guid A1, version X1}, {guid A2, version X2}, ...}}
  {application B, {{guid B1, version Y1}, {guid B2, version Y2}, ...}}
  ...
}

```

Matching the lists will happen on two levels. The outer loop, similar to the one above, will match the application names. If the names match, then another sub-loop will compare the guids of the existing and

incoming policies. The only caveat that needs to be kept in mind is that the next step after each application is processed in the loop is actually merging the policies on a per-application basis. The logic above assumes that the list is sorted by GUIDs, but merging requires sorting by precedence. This can be overcome in different ways:

- Using the sorting precedence sorting of the policies and no use the optimized algorithm for processing policies on per application basis
- Keep two indexes
- Resort

The final determination of which logic to use will be made during the implementation phase, based on the complexity of the code and performance observations.

Lookup Actions

Looking up actions is similar to dealing with the policies on a per-application basis. The difference is that there is no need to merge the policies after they are received, but rather execute them according to the precedence order. It is important to take into account that only new or updated actions will be executed. The merging logic that is presented above might not be the best approach in this case, because it would require resorting. Alternative or modified solutions will be considered during the implementation phase.

XML-RPC Interface

The XML-RPC interface will expose entry points that will perform following operations:

- Get Policy
 - Get Policy by GUID - returned policy is a compressed XML blob
 - Get Policy by GUID - returned policy is uncompressed XML blob
 - Get Policy by GUID encrypted with user provided password - returned policy is compressed XML blob encrypted with hash of the password.
 - Get Policy by Application - returns list of configuration policies related to an application in precedence order
- Update Policy
 - Update policy by GUID - creates a new version of the policy
- Add Policy
 - Add Policy - adds a new instance of the policy to the policy store
- Delete Policy
 - Delete Policy by GUID - deletes specified policy
 - Delete Policy by application - deletes all policies for specified application
- Assign Policy
 - Assign policy - replaces current list of hosts and host groups the policy is assigned to with a provided list
- Apply Policy
 - Apply Policy by GUID - currently saved version of the policy is applied
 - Rollback Policy by GUID - undo last applied policy

XML-RPC Connection

The XML-RPC connection uses SSL. This SSL is not mutual authentication. The client will authenticate to the server using Kerberos over GSSAPI. The client will authenticate the server by using the public key from the server certificate. The only problem with this arrangement is the provisioning of the certificate to all client machines. Requiring that the certificate be copied from the IPA server to the client during the client installation process seems like a bad approach, and a major blocker for deploying the IPA solution as a whole. Delivering the certificate to the client so that the XML-RPC interface can work needs to be done using an LDAP channel.

To solve this problem, a special entry will be created in the configuration area of DS's DIT. This entry will contain the IPA server's public certificate and its version. It will be populated automatically when the primary server is installed. Each time the client connects to DS it will check if the certificate version it has is older than the one in the DS. If the DS has a newer version of the certificate, or if the client does not have a

certificate at all, then the client will download the certificate and store it for use in the XML-RPC connection. The Policy Downloader will be blocked until a valid certificate is delivered to the client by the IPA data provider. In the rare case where the customer might want to replace his server certificate, he would have to generate a new certificate and key pair, then update the above-mentioned configuration entry with the new certificate and update the version.

Developing and Testing Policies

It is expected that customers will have a test IPA realm where they will test their environment. IPA V2 plans to provide a set of scripts that will allow customers to dump a policy they have developed in a test environment and then reload it into a production server. It is clear that in order to migrate a policy correctly from a test IPA domain to a production IPA domain, some entries from the test DS also need to be carried forward. We plan to provide a set of scripts that will ease solving this migration use case. The scope and complexity of such scripts will be assessed later.

Policies for the IPA Client

Policies for the IPA client will control the behavior of the IPA client itself. So far, the following configuration objects have been considered for inclusion in the IPA Client configuration policy:

- should the IPA client allow access for IPA users when the system is offline?
- the cache lifetime for different LDB entities
- how frequently policies should be downloaded from the server

etc.

The IPA client will be installed with the default values pre-populated in the LDB, but the centralized policy would be able to alter this data on different subsets of hosts by applying specific IPA client policies to specific hosts.

DS Schema for Policy Related Objects

Schema for the policy related objects is described on the following page. <TBD>

Adding Support for New applications

Adding New Application Roles

If one wants IPA to support role management for a new application that IPA does not yet know about, it will be necessary to conduct the following development steps:

- develop a Relax NG schema for the role definition policy following the roles described on this page. A more detailed guide will be developed as part of the documentation set.
- drop this new schema into a schema storage area on all IPA server replicas. This is a manual step.
- develop XSLT templates that will be run on the clients
- deliver XSLT templates and schema definitions to the clients using one of:
 - IPA's Actions mechanism
 - Satellite
 - manual delivery
 - some other delivery mechanism
- add a new application entry to the IPA database. Let it replicate.
- start a new administrative session

At this point the system is ready to define or import new roles.

Adding New Configuration Policies

If one wants IPA to support configuration management for a new application that IPA does not yet know about, it will be necessary to conduct the following development steps:

- develop a Relax NG schema for the role definition policy following the roles described on this page. A more detailed guide will be developed as part of the documentation set. Keep in mind that the schema should include all required metadata that would control the merging logic on the client.
- drop this new schema into a schema storage area on all IPA server replicas. This is a manual step.
- develop XSLT templates that will be run on the clients
- develop local file merge utilities if needed
- deliver XSLT templates, schema definitions and merge utilities to the clients using one of:
 - IPA's Actions mechanism
 - Satellite
 - manual delivery
 - some other delivery mechanism
- add a new application entry to the IPA database. Let it replicate.
- start a new administrative session

At this point the system is ready to define or import new configuration policies.

Retrieved from "https://wiki.idm.lab.bos.redhat.com/export/idmwiki/Overall_Design_of_Policy_Related_Components"

- This page was last modified on 29 January 2009, at 04:44.