Policy Discription Language Module

Requirements & design

G.M. Venekamp
venekamp@nikhef.nl

September 23, 2003

Abstract

When a user or service has been authenticated and is allowed into the system, the system must provide the user or service with a suitable environment. In order to decide which environment is correct, the user or service needs to provide a set of credentials. Based upon these credentials, the system assigns the allowed resources.

In order to accomplish the above, the system must know about policies for assigning resources to a user or service. To be as flexible as possible, a configuration file is used for describing policies. A small and simple yet powerful language has been developed to describe the policy rules of a site. This document describes the policy description language design.

document version: 0.1

Document history:

Date	Version	Author	Change
10 February 2003	0.1	G.M. Venekamp	Initial document.

Contents

1 Introduction **1**

2 Requirements **3**

3 Existing solutions **5**

3.1 Boolean expressions 5

3.1.1 Policies in a Generic AAA Environment 5

3.1.2 A Policy Description Language 5

3.2 PAM 6

4 Design **7**

4.1 DFA example 7

4.1.1 Policy example 8

4.2 The language 8

4.2.1 Policy example 9

4.2.2 More elaborate DFAs 10

4.3 Extending the language 10

4.3.1 Policy example 12

4.4 Final words 12

1 Introduction

When a user or server wants to make use of a grid enabled computer, it first needs permission to enter the grid enabled service. After it has been determined that the user is allowed onto the system, it must be decided what the user is allowed to do and what to use. The right environment must be set up. In other words, the system must be able to determine what resources are available to a particular user. The system is aided by the fact that each user carries a set of credentials. These credentials describe what a user is allowed to do. For example, the credentials tell of which virtual organization (VO) a user is a member. The user can selected anyone of these VOs to be used on his account. With this information the system is able to tell how much disk space may be used, how many processors are reserved for the VO, billing, etc. The system must consult these credentials in order to determine the environment.

Having credentials is not enough. A site might want to have control of the way the environment is setup. Consider the following: A user might be a member of a VO that has access to your site. At the same time this user has a local account on your site. When this user want to use some of the resources of the site, either it is done on the account of the VO or as a local user of that system. A site has a choice now. When a user with a local account wants to make use of the resources, a site can decide that the user can use the resources under his/her own account regardless of any VO he/she is a member of. Of course a site might also decide that even though a user has a local account, he/she needs to use the VO affiliation first.

Is dit wel een goed voorbeeld?

For this reason, each site has a set of policies. These policies, together with the credentials, determine the environment of the user. The policy rules are described in a configuration file. This configuration file is composed of plain text to make it readable by humans and editable by a wide range of editors. Another important fact is the readability of the file. This should be is as intuitive as possible. If the description of policies is too difficult, errors are easily made. Mistakes can lead to assigning resources to a person which has no right to that particular resource. It would be embarrassing to send the billing to the wrong VO. Clearly, this must be prevented as much as possible. By trying to keep the language a simple as possible, it is our believe that these kinds of mistakes should be kept to a minimum.

Also, by not over complicating the structure of the configuration file, graphical user interfaces can be easily created. Thus, further limiting unwanted mistakes. This is, however, not the first goal at this stage of development, but it might be desirable once the subsystem is in use by less experienced users.

Structure of the document

The following section describes the requirements of the language. Requirements help us understand what it is we really want. This is followed by a discussion on existing solutions and their usability to the project. Since non of the existing solution fulfill our needs, we have to come up with our own language. Section 4 shows the design of that language.

2 Requirements

pdl_1: The configuration file must be in a human readable format.

Rationale: When the configuration file is in a human readable format, it can be edited in any plain text editor by any human. This relinquishes the needs for a GUI to edit configuration files and can be done quite easily remotely.

pdl_2: The configuration file must support comments.

Rationale: The configuration file becomes more readable by adding comments. This should lead to a better interpretation of the file and reduce the number of errors made.

pdl_3: The configuration language must support for more than one policy rule.

Rationale: By allowing several smaller policy rules, instead of one huge rule, the readability of the configuration file is enhanced and thereby reducing the number of errors.

pdl_4: A policy must be reusable inside another policy.

Rationale: By allowing policy rules to contain policy rules, one reduces the amount of double policy rules. Thus limiting the number of errors. Should an error be discovered and fixed, it is fixed for all policy rules. No fixes are forgotten this way.

pdl_5: The language must support both a TRUE and FALSE path, when a part of the policy rules fails.

Rationale: Policy modules return either TRUE or FALSE. When a module succeeds or fails, two different paths can be taken to complete the policy. The language must reflect this behaviour.

3 Existing solutions

The language that is going to be used in the configuration files to describe policy rules must be readable to a human and not too difficult to grasp its meaning. To make it readable means to use the plain text character set. It can be read by a large variety of editors across all platforms. When remote administration is needed, the plain text character set insures the best compatible solution.

3.1 Boolean expressions

One of the approaches that have been considered are boolean expressions. All requirements are met. However, there is one drawback of using boolean expressions; they get more complicated as the length of the expression grows. For example, the boolean expression:

$$q_1 \wedge (q_2 \vee q_3)$$

is quite easy to interpret and understand. The story is all together quite different for the following expression:

$$q_1 \wedge q_2 \wedge ((q_3 \wedge q_4) \vee ((q_5 \vee (q_6 \wedge q_7)) \wedge q_8))$$

Therefore we have decided not to use boolean expressions as our configuration language.

3.1.1 Policies in a Generic AAA Environment

In [2], a generic policy language is described. In essence, the language is still a boolean expression. The basis of the language is that of the form: '**if** *condition* **then** *action*', where the condition is a boolean expression in k-DNF form. For our purpose, the action is of less interest to us. For us, it is the modules (a tuple from the k-DNF expression) that performs the action and at the same time provide the condition. To conclude, we did not choose this solution because:

1. boolean expressions are the core of the language, making it harder to read;
2. the '**if** *condition* **then** *action*' does not map to our domain as well as we would like.

3.1.2 A Policy Description Language

In [1], Lobo describes yet another approach to a Policy Description Language. This is much the same as A. Taal does in: Policies in a Generic AAA Environment. This approach is less expressive. Therefore based on the same reasons as mentioned above (section 3.1.1), this approach is not pursued any further.

3.2 PAM

As stated in the previous section, using a human readable format is not enough. The language itself must also be easy to interpret by a human. One such solution is PAM (Pluggable Authentication Modules). Within PAM the success of a module is either: required, optional or sufficient. The meaning of each conditional is the following:

required – this module must return success in order to have the overall result be successful;

optional – if this module fails the overall result can still be successful if another module in the stack returns success;

sufficient – if this module is successful, skip the remaining modules in the stack, even if they are labeled as required.

This is in conflict with requirement pdl_5 on page 3. When it is sufficient for a module to succeed, PAM exits and returns TRUE to indicate that it has successfully exited. One cannot, based on the success of the sufficient module, evaluate additional modules. Only on the failure of the module, will PAM continue with the next listed module on the stack.

4 Design

Our solution is to envision the policy rules as a Deterministic Finite Automata (DFA). DFAs are composed of states and transitions from one state to another. Precisely one of the states of a DFA has to be a starting state. This state is identified by an arrow, with a filled disk attached to the beginning, pointing to the start state. There is always exactly one start state per DFA. Also, a DFA has at least one end state. These are drawn with an inner circle. Unlike start states, a DFA can have more than one end state. Transitions are drawn as arrows originating from, and pointing to, a state. Transitions only happen when the conditions for the transition have been met. When no more transitions can be made, the DFA is said to have been terminated. If it terminates in an end point the DFA has succeeded, if not, it has failed.

Normally, a transition can be anything. In our case however, we know that there are only two possible transitions: TRUE or FALSE. We can use this a priori knowledge in designing our language, i.e. we do not have to specify the transition type; it is either TRUE or FALSE. Also with regard to termination of the DFA, we do it slightly different. When the DFA reaches an end state, the evaluation of that state determines the success or failure of the DFA. This is in contrast to reaching an end state and reporting success. One can view this difference as a compression. This way one less state needs to be drawn.

klopt dit gramaticaal wel?

4.1 DFA example

The following figure illustrates a DFA in which three states $\{q_1, q_2, q_3\}$ need to be visited. In each state a different module is evaluated and the result is used for the transition. Since we are only interested in successful evaluations, the transition only takes place when a module returns TRUE. Given the following DFA: we can say that the policy rule

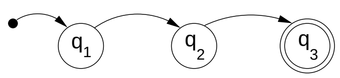


Figure 1: Simple DFA containing three states and only one TRUE transitions.

has succeeded when the module in state q_3 returns TRUE. Should the result be FALSE for the states $\{q_1, q_2\}$ then the policy rule has failed. When no transition can be made, the DFA terminates in that state and the policy rule has failed.

Of course a FALSE transition is also possible. We do need to make clear that a transaction needs to take place on the FALSE condition. Therefore we introduce the letters T and F to denote TRUE and FALSE respectively. The following figure shows a DFA containing a FALSE transition. The way this DFA should be read is as follows: q_1 must succeed and either q_2 or q_3 must succeed.

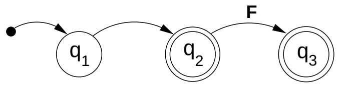


Figure 2: Simple DFA containing three states, two TRUE and one FALSE transitions.

In case the evaluation of q_2 is TRUE, we do not need to evaluate q_3, because q_2 is marked as an end state. The policy rule has succeeded with the successful evaluation of q_2. Only when q_2 returns FALSE, is the transition from q_2 to q_3 made and determines q_3 the success or failure of the policy rule.

4.1.1 Policy example

Figure 3 shows a real example of a policy. Sites which wish to give precedence to localaccounts over poolaccounts might want a policy like this. The policy checks if the user may use a localaccount first. If this is allowed, the policy then checks for posix enforcement. When all is okay, the policy has succeeded, otherwise it has not. However, it is possible that the localaccount does not allow for a local account because, the user has no local account at all. In this case the site policy tells that the poolaccount needs to be checked. When the poolaccount is successful, the voms will be checked. At last, the same posix enforcement need to be done as before. Only when the posix enforcement is successful, will the policy have succeeded.

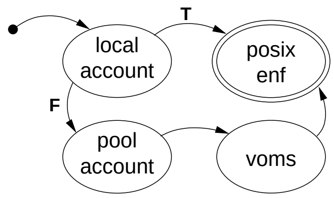


Figure 3: Real example.

In the above example the DFA starts at the localaccount. By letting it start at the poolaccount state and inverting the direction of the transition between localaccount and poolaccount, precedence is given to poolaccount instead of localaccount. Thus, even when a user has a local account, the system prefers pool account. This shows how system administrators are able to decide what happens on their system.

4.2 The language

DFAs give us a reasonable clear language to express policy rules. However, this is in direct violation of requirement pdl_1. We need to translate the DFA into plain text.

In order to describe the DFA, we only need to specify the transitions from one state to another. We use the '$\rightarrow$' symbol to indicate a transition. Thus $q_1 \rightarrow q_2$ means the transition from state q_1 to q_2. From this notation it is not clear what the condition of the transition is. Unless stated otherwise, the default condition of a transition is TRUE. With these rules, the DFA of Figure 1 translates into:

$$\begin{array}{rcl} q_1 & \rightarrow & q_2 \\ q_2 & \rightarrow & q_3 \end{array}$$

If the DFA contains a FALSE transition, then we need to tell that we do *not* want to use the default TRUE transition. This is accomplished by prefixing a tilde to a transition line. Thus, Figure 2 translates into:

$$\begin{array}{rcl} q_1 & \rightarrow & q_2 \\ \sim q_2 & \rightarrow & q_3 \end{array}$$

This kind of language is easily translated into plain text. The '$\rightarrow$' symbol can be written in plain text as: '`->`'. For typographically reason we will use the '$\rightarrow$' symbol throughout this document. The language just describes can also be written down into Extended Backus Naur Form (EBNF). We use the following definitions:

- '$\star$' denotes zero or more repetitions;
- '+' denotes one or more repetitions;
- '|' denotes a choice;
- anything between quotes (') is taken as literal;
- reserved words are printed in bold.

Using the above definitions, we can express our language as follows:

$$\begin{array}{rcl} policy & ::= & rule+ \\ rule & ::= & term \ \rightarrow term \\ rule & ::= & term \ \rightarrow term \ '|' \ term \\ rule & ::= & \sim term \ \rightarrow term \\ term & ::= & [\text{a–zA–Z0–9\_.}]\star \end{array}$$

The above reads: a policy contains at least one rule; a rule can be one of three forms; a term is composed out of characters, digits, underscores and punctuations.

4.2.1 Policy example

With the above definitions, the policy as described in § 4.1.1 is thus written:

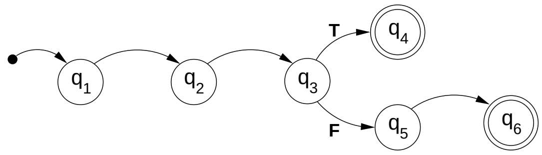
$$\begin{array}{rcl} localaccount & \rightarrow & posix\_enf \mid poolaccount \\ poolaccount & \rightarrow & voms \\ voms & \rightarrow & posix\_enf \end{array}$$

As one can see, this is fairly simple to interpret and understand. It also meets all requirements we have set ourselves.

We would like more flexibility in the language. Thus, the basic idea will be extended. First, we will show some more complex artificial examples to demonstrate the ease of the language.

4.2.2 More elaborate DFAs

We will give two more examples of a DFA. This is to show that rather complicated policies can be written down in an understandable fashion. Suppose we have the following DFA:

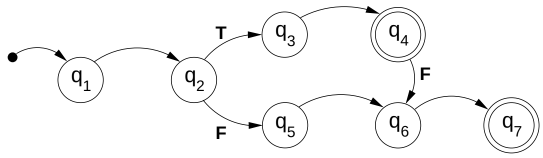


This would translate into:

$$\begin{array}{lcl} q_1 & \rightarrow & q_2 \\ q_2 & \rightarrow & q_3 \\ q_3 & \rightarrow & q_4 \mid q_5 \\ q_5 & \rightarrow & q_6 \end{array}$$

Even though the DFA has six states, the configuration file does not look overwhelming. Which is what we are looking for.

As a last example we give the following DFA:



Again, this is translated into the following policy rule:

$$\begin{array}{rcl} q_1 & \rightarrow & q_2 \\ q_2 & \rightarrow & q_3 \mid q_5 \\ q_3 & \rightarrow & q_4 \\ \sim q_4 & \rightarrow & q_6 \\ q_5 & \rightarrow & q_6 \\ q_6 & \rightarrow & q_7 \end{array}$$

Still, the configuration file looks comprehensible despite the fact that several TRUE/FALSE transitions are present.

4.3 Extending the language

What we have described thus far is a very basic language. It does all that we need. Though, we would like to take it one step further. First of all, we would like to be able to use free variables. This allows us to use simpler naming in the policy rules. A free variable is defined as follows:

$$\textit{var} \ ::= \ \text{'='} \ \textit{term}$$

However, spaces are not allowed in this form, since a term cannot contain a space. If spaces are needed, the term needs to be a string:

$$\textit{var} \ ::= \ \text{'='} \ \textit{string}$$

A string is defined as the following regular expression:

$$\textit{string} \;::=\; \backslash"[\hat{}\backslash"\backslash n]\star[\backslash"\backslash n]$$

The string matches any sequence that starts with a double quote, followed by any character that is not a newline and closed by a double quote.

Sites might want to have more than one policy rule. Thus, we add the following line to the definition:

$$\textit{config} \;::=\; \textit{var}\star\ \textit{policy}\text{':'}+$$

This gives every policy rule a label, thereby grouping them and being able to refer to each individual group. Now that policies are labeled we would like to have the ability to use a policy rule inside a policy rule. This means that a 'term' can also be a 'policy':

$$\textit{term} \;::=\; \textit{policy}$$

All modules are stored at the same location. Therefore we have reserved one name: **path**. This is a special variable. It needs to be followed by the assignment character (=) and a Unix path.

$$\textit{var} \;::=\; \textbf{path}\ \text{'='}\ \textit{pvar}$$

The following regular expression defines paths:

$$\textit{pvar} \;::=\; \text{'='}\ [\backslash/\backslash.][\hat{}\backslash/\ \backslash t\backslash n]\star$$

Last but not least, we would like comments as well. A comment is defined in the following manner:

$$\textit{comment} \;::=\; \text{'\#'}\ \{\textit{any\_character}\}\ \text{'\textbackslash n'}$$

Given our original specification together with the above extensions, the complete EBNF notation for the language is:

$$\begin{array}{rcl}
\textit{config} & ::= & \textit{var}\star\ \textit{policy}\text{':'}+ \\
\textit{var} & ::= & \text{'='}\ \textit{term} \\
\textit{var} & ::= & \text{'='}\ \textit{string} \\
\textit{var} & ::= & \textbf{path}\ \text{'='}\ \textit{pvar} \\
\textit{policy} & ::= & \textit{rule}+ \\
\textit{rule} & ::= & \textit{term}\ \rightarrow \textit{term} \\
\textit{rule} & ::= & \textit{term}\ \rightarrow \textit{term}\ \text{'|'}\ \textit{term} \\
\textit{rule} & ::= & \text{\textasciitilde}\textit{term}\ \rightarrow \textit{term} \\
\textit{term} & ::= & [\text{a–zA–Z0–9_.}]\star \\
\textit{comment} & ::= & \text{'\#'}\ \{\textit{any_character}\}\ \text{'\textbackslash n'}
\end{array}$$

4.3.1 Policy example

Given the definition of the language in the previous paragraph. We can write a configuration file for the example in Figure 3 on page 8:

```
1  #  Configuration example in which local accounts are
2  #  preferred over pool accounts.
3
4  #  First we define the path where the modules can be found.
5  path = /opt/edg/lib/lcmaps/modules
6
7  #  Let us now define the variables.
8  local = "lcmaps_localaccount.mod -gridmapfile /etc/grid-security/grid-mapfile"
9  pool  = "lcmaps_poolaccount.mod  -gridmapfile /etc/grid-security/grid-mapfile"
10 voms  = "lcmaps_voms.mod         -vomsdir /etc/grid-security/certificates
11                                  -certdir /etc/grid-security/certificates"
12 posix = "lcmaps_posix.mod        -maxuid 1 -maxpgid 1 -maxsgid 32"
13
14 #  We have one policy rule that is called default.
15 default:
16 local -> posix | pool  # See of local and posix do the job.
17 pool  -> voms          # If not: pool, voms and posix
18 voms  -> posix         # need to do it.
```

The first interesting line to look at is line 5, we see the use of the reserved word **path**. Here the path is defined where the modules can be found. Lines 8 – 12 define four variables. Each variable is a module with its arguments. These variables are used in the policy rules as can be seen on lines 16 – 18. Line 15 defines the name of the policy.

4.4 Final words

As one can see, policies in the graphical form of the DFA are quite easy to comprehend. On the other hand, translating from the graphical from to plain text is quite easy to do and not error prone. The resulting configuration file is still easy to interpret and understand. Mistakes completely eradicated. However, mistakes are less likely to be made by drawing a DFA and then translating it. As a bonus, one can easily create a GUI for creating and maintaining the policy rules.

References

[1] Jorge Lobo, Randeep Bhatia, and Shamim A. Naqvi. A policy description language. In *AAAI/IAAI*, pages 291–298, 1999. `http://citeseer.nj.nec.com/lobo99policy.html`.

[2] A. Taal and et.al. Policies in a generic aaa environment. `http://www.aaaarch.org/doc08/taal-aaaarch-generic-pol-00.txt`, November 2000.