NGAC policy tool and policy server v0.3.3

Summary of v0.3.3

This version of the TOG NGAC Policy Tool and Policy Server includes new features that represent the merging of requirements from several use cases. The implementations of these features are in various stages of completion in this version. The new features and their current status in this version are:

- Ability to provide a text description of the current version, settable in the param.h file. This is used to provide a more specific description to characterize the current version, whether it is a release or a development version. This is reflected in the behaviour of the ***version*** and ***versions*** commands of the Policy Tool.

- ***'all'*** composition and modified ***setpol*** API – a distinguished "current policy" setting in the Server that invokes a specific composition of all of the subsequently loaded policies. This mode of policy composition is selected by invoking the ***setpol*** API with the policy identifier 'all', and acts over all subsequently loaded policies until another ***setpol*** API call cancels it. A call to the ***setpol*** API with this argument clears the policy information point in the server and sets the current policy to this distinguished name. A subsequent call to ***setpol*** with any other policy name will set the current policy to the policy identified in the argument and will cancel the 'all' composition.

- ***composed_policy*** declaration in the policy specification language – The implementation accepts a declaration that a new policy is to be created from the composition of two previously defined policies. The construct is accepted as part of the language but the performance of the composition is not yet implemented internally. Note that it is out of place for ***composed_policy*** to appear within a policy elements list. Consideration is being given to either a) allowing ***composed_policy*** as an alternative to the policy elements list in a ***policy*** declaration, or b) to allow ***composed_policy*** as an alternative to the ***policy*** declaration. User feedback is welcomed.

- ***operation*** declaration in the policy specification language – The implementation accepts declarations of operations. The declaration may appear with one or two arguments. The first argument is an *operation identifier* (operator), the optional second argument is *operation information* that provides more information about how the operator can be used. It is intended to provide the basis for type checking association and access request arguments. The construct is accepted as part of the language but the information is not yet used internally to perform checks.

- ***opset*** declaration in the policy specification language – The implementation accepts declarations of operation sets. The declaration has two arguments, an *operation set identifier* and a *list of operations*. The operation set identifier may be used in an association declaration in lieu of an operations list. Note that an opset is not the same as a type for an object, which includes all operations that are associated with an object type. An opset introduces a name that represents a particular list of operations. Future **object_type** declarations will also accept operation set identifiers, in which case the specified operation set will be interpreted as the set of all operations that are associated with the object type. The current version accepts ***opset*** declarations but does not yet use them.

- Extensions to the Policy Administration API – in particular, ***load***/***unload*** as synonyms for ***importpol***/***purgepol*** (which are deprecated names), and the required ***token*** HTTP parameter, details follow.

- Extensions to command line options handling in the Policy Server – in particular, ***permit***/***grant***, ***deny***, and ***token*** options, and some added synonyms for options, details follow.

- The ***mkngac*** script compiles separate executables for the Policy Tool and the Policy Server. Using the server executable is the only practical way to access the server's command line options.

Enhanced declarative policy specification language

The enhanced declarative policy language supports policy composition and the definition of new object classes and operations.

A declarative policy specification is of the form:

policy(<policy name>, <policy root>, <policy elements>).

where,

<policy name> is an identifier for the policy definition

<policy root> is an identifier for the policy class defined by this definition

<policy elements> is a list [*<element>, ... , <element>*]

where each *<element>* is one of:

user(<user identifier>)

user_attribute(<user attribute identifier>)

object_class(<object class identifier>, <operations>)

object(<object identifier>)

object(<object identifier>, <object class identifier>, <inh>, <host name>,
<path name>, <base node type>, <base node name>)

object_attribute(<object attribute identifier>)

policy_class(<policy class identifier>)

composed_policy(<new policy name>, <policy name1>, <policy name2>)

operation(<operation identifier>)

assign(<entity identifier>, <entity identifier>)

associate(<user attribute id>, <operations>, <object attribute id>)

where <*operations*> is a list:

[<*operation identifier*>, ... , <*operation identifier*>]

connector('PM')

The initial character of all identifiers must be a lower-case letter or the identifier must be quoted with single quotes, e.g. *smith* or *'Smith'* (identifiers are case sensitive so these examples are distinct). Quoting of an identifier that starts with a lower-case letter is optional, e.g. *smith* and *'smith'* are not distinct.

Additionally:

< *inh* > can be **yes** or **no**.

< *host name*> contains the name of the host where the corresponding file system object resides.

< *path name*> is the complete path name of the corresponding file system object.

Enhanced ‘ngac’ policy tool

The ‘ngac’ policy tool for doing standalone policy development and testing is extended for policy composition and with the ability to start the security server. The tools offers the command line prompt “ngac>”, to which it accepts the following admin user commands:

- `access(` <policy name>`,` <permission triple>`).`
  Check whether a permission triple is a derived privilege of the policy.

- `admin.`
  Switch to admin (normal) user mode.

- `advanced.`
  Switch to advanced user mode.

- `aoa(` <user>`).`
  Show the user accessible object attributes of the current policy.

- `combine(` <policy name 1>`,` <policy name 2>`,` <combined policy name>`).`
  Combine two policies to form a new combined policy with the given name.

- `echo(` <string >`).`
  Print the argument string, useful in command procedures.

- `halt.`
  Exit the policy tool. (Will also terminate spawned server.)

- `help.`
  List the commands available in the current mode.

- `help(` <command name>`).`
  Give a synopsis of the named command.

- `import_policy(` <policy file>`).`
  Import a declarative policy file and make it the current policy.

- `newpol(` <policy name>`).`
  Set the named policy to be the new current policy.

- `nl.`
  Print a newline, useful in command procedures.

- `proc(` <procedure name>`[, step]).`
  Run the named command procedure, optionally pausing after each command.

- `proc(` <procedure name> [`, verbose`] `).`
  Run the named command procedure, optionally verbose.

- `regtest.`
  Run built-in regression tests.

- `script(` <file name> [`, step`] `).`
  Run the named command file, optionally pausing after each command.

- `script(` <file name> [`, verbose`] `).`
  Run the named command file, optionally verbose.

- `selftest.`
  Run built-in self tests.

- `server(` <port > `).`
  Start the policy server on the given port number.

- `server(` <port >`,` <admin token> `).`
  Start the server on the given port number, with given admin token.

- `version.`
  Display the current version number and version description.

- `versions.`
  Display past and current versions with descriptions.

There are, and may in the future be, advanced user commands for development and diagnostics.

Enhanced 'ngac-server' policy server

Comprising the Policy Decision Point (PDP), the Policy Administration Point(PAP), and the Policy Information Point (PIP) of the NGAC functional architecture. The policy server implements a Policy Query Interface API to be queried by PEPs, and a Policy Administration Interface API to be used to incrementally change the policy the server is using to compute access queries.

The policy server may be initiated within the 'ngac' tool by issuing the command `server( <port> ).` or the command `server( <port>, <token> ).` at the tool's command prompt "ngac>". The preferred way to initiate the server in a production environment is by using the compiled executable, which makes the command line options available.

Policy Query Interface

The enhanced server offers the following RESTful APIs as the Policy Query Interface:

pqapi/access – test for access permission under current policy

Parameters

- user = <user identifier>
- ar = <access right>
- object = <object identifier>

Returns

- "permit" or "deny" based on the current policy
- "no current policy" if the server does not have a current policy set

Effects

- none

pqapi/getobjectinfo – get object metadata

Parameters

- object = <object identifier>

Returns

- "object=<obj id>,oclass=<obj class>,inh=<t/f>,host=<host>, path=<path>,basetype=<btype>,basename=bname>"

Effects

- none

Policy Administration Interface

The enhanced server offers the following APIs as the Policy Administration Interface. A "failure" response is typically preceded by a string indicating the reason for the failure.

All of the APIs of the Policy Administration Interface have an additional final parameter not shown in the Parameters lists below, that is, a ***token*** parameter that acts as a key to use the interface. The provided token must match the token used when the Server was started, or 'admin_token' (the default) if no token was specified at startup.

paapi/getpol – get current policy being used for policy queries

Parameters

- none

Returns

- <policy identifier> or “failure”

Effects

- none

paapi/setpol – set current policy to be used for policy queries

Parameters

- policy = <policy identifier>

Returns

- “success” or “failure”
- “unknown policy” if the named policy is not known to the server

Effects

- sets the server’s current policy to the named policy
- the distinguished policy name “all” causes the composition of all loaded policies to be applied

paapi/add – add an element to the current policy

Parameters

- policy = <policy identifier>
- policyelement = <policy element> only user, object, and assignment elements as defined in the declarative policy language; restriction: only user to user attribute and object to object attribute assignments may be added. Elements referred to by an assignment must be added before adding an assignment that refers to them.[1]

Returns

- “success” or “failure”

Effects

- The named policy is augmented with the provided policy element

paapi/delete – delete an element from the current policy

Parameters

- policy = <policy identifier>
- policyelement = <policy element> permits only user, object, and assignment elements as defined in the declarative policy language; restriction: only user-to-user-attribute and object-to-object-attribute assignments may be deleted. Assignments must be deleted before the elements to which they refer.

Returns

- “success” or “failure”

Effects

- The specified policy element is deleted from the named policy

paapi/combinepol – combine policies to form new policy

Parameters

[1] In a purely declarative sense this restriction would not be necessary. This applies also to the restriction on ***delete***. It is necessary only because at the time that the ***add*** is encountered the implementation performs a check to guarantee that the resulting policy will not have “loose ends”. A different implementation could perform a consistency check before a policy is used that has had ***add***/***delete*** operations performed since it was last checked.

- policy1 = <first policy name>
- policy2 = <second policy name>
- combined = <combined policy name>

Returns
- "success" or "failure"
- "error combining policies" if the combine operation fails for any reason

Effects
- the new combined policy is stored on the server

paapi/load – load a policy file into the server

Parameters
- policyfile = <policy file name>

Returns
- "success" or "failure"

Effects
- stores the loaded policy in the server
- does NOT set the server's current policy to the loaded policy

paapi/unload – unload a policy from the server

Parameters
- policy = <policy name>

Returns
- "success" or "failure"
- "unknown policy" if the named policy is not known to the server

Effects
- the named policy is unloaded from the server
- sets the server's current policy to "none" if the unloaded policy is the current policy

Sessions in the Policy Server

An active session identifier may be used as an alternative to a user identifier in an access query made to the Policy Query Interface.

paapi/initsession – initiate a session for user on the server

Parameters
- session = <session identifier>
- user = <user identifier>

Returns
- "success" or "failure"
- "session already registered" if already known to the server

Effects
- the new session and user is stored

paapi/endsession – end a session on the server

Parameters
- session = <session identifier>

Returns
- "success" or "failure"
- "session unknown" if not known to the server

Effects
- the identified session is deleted from the server

Policy Server command line options

When a compiled version of the policy tool or the policy server is started from the command line, several command line options are recognized.

- `--token`, `-t` <admintoken> use the token to authenticate requests to the paapi (formerly `--admin` or `-a`) (not yet implemented)
- `--deny`, `-d` respond to all access requests with `deny`
- `--grant, --permit`, `-g` respond to all access requests with `grant`
- `--import`, `--load, --policy, -i, -l` <policyfile> import/load the policy file on startup
- `--port`, `--portnumber, --pqport, -p` <portnumber> server should listen on specified port number
- `--selftest`, `-s` run self tests on startup
- `--verbose`, `-v` show all messages

Policy Composition

The policy server supports two forms of policy composition. The first is achieved with the ***comebinepol*** API. It forms the composition of policies as described in the NGAC literature and examples.

The ***'all'*** policy composition is a distinct form of policy composition. When the policy servers current policy is set to 'all' through the ***setpol*** API, all currently loaded policies are automatically combined for every ***access*** request. The manner in which the polices are combined is as follows:

- Every policy is first qualified to participate in computing the verdict of an ***access*** request. There are several subtle variations possible for qualification, and which to use is a parameter of the system. With the current setting of this parameter in the param module (all_composition = p_uo) to qualify a policy must be defined to have explicit jurisdiction over both the user and the object specified in the ***access*** request. There must be at least one qualifying policy.
- All qualified policies are queried with the triple (user, access right, object) specified in the ***access*** request. If any qualified policy returns 'deny' then the ***access*** request returns 'deny'.

Sets of policies to be combined according to the 'all' policy composition should be designed with the foregoing runtime semantics taken into

consideration. The selection of the p_uo variant is currently a non-modifiable system parameter. It is being considered whether to permit the selection of the variant of the ***all_composition*** parameter to be specified when the 'all' composition mode is activated in the server through the ***setpol*** call.

Protecting the Policy Administration Interface

The policy administration functions should not be made available to the normal object PEPs. Rather the Policy Administration API should be accessible only to an administratively authorised user through the policy administration tool or a process with the same authorisation, and some functions such as ***setpol***/***getpol*** should be accessible only to the "shell" program that executes the NGAC client application. In this way, the "shell" that controls execution of the application would also determine the user/session and policy under which the application should execute.

These protections may be achieved by appropriate use of the host operating system features, if such features are available. For example, on a Unix-like system, domain-specific trusted "shell" programs can be ***setuid*** to the owner of the domain, with the associated privileges passed along over a fork call and revoked before the child process performs an exec of an untrusted application program.

The admin token should be generated by the top-level process and passed in the ***token*** option when it starts the Policy Server. It or another trusted process that it spawns, to which it passes the token, can use the token to perform policy administration calls to the Server.

If the Policy Server is started without the ***token*** option it will use the default admin token defined in the param module in param.pl. The default is 'admin_token'. This default can be used in a benign environment or for development and testing. Note however that in a production environment where the Policy Administration Interface is not protected an untrusted process would be able to manipulate the policy being used by the Server.