NGAC policy tool and policy server Release Note for v0.3.4+++, 11 October 2019

1. SUMMARY OF V0.3.4+[1]

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

[1] This document describes the development version v0.3.4+. Some of the features are new and not exhaustively tested and some features specified in this document have not yet been completely implemented.

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. Also, ***loadi*** and ***readpol*** are new APIs that have been added.

- 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 way to access the server's command line options.

2. THE NGAC POLICY TOOL AND POLICY SERVER

The NGAC functional architecture presented in the standard is shown in Figure 1. PEP is Policy Enforcement Point, RAP is Resource Access Point, PDP is Policy Decision Point, PAP is Policy Access/Administration Point, PIP is Policy Information Point, and the optional EPP is Event Processing Point.

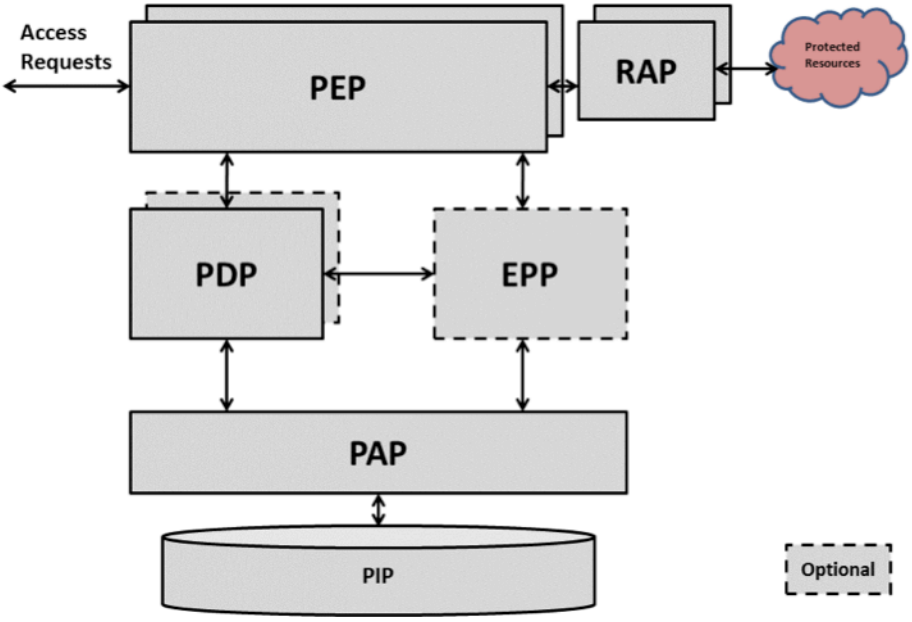


Figure 1 NGAC functional architecture per the standard

The Open Group's version of the functional architecture differs from the version presented in past NGAC documents and standards in that it "unbundles" the PEPs and RAPs from the NGAC perimeter as shown in Figure 2. This is a response to our practical experience with getting application developers to adapt their code to use NGAC to access their resources. It is not practical to modify the NGAC implementation every time it is desired to add new protected object class access methods. This architecture enables a more extensible implementation of NGAC by easing the addition of new protected object kinds.

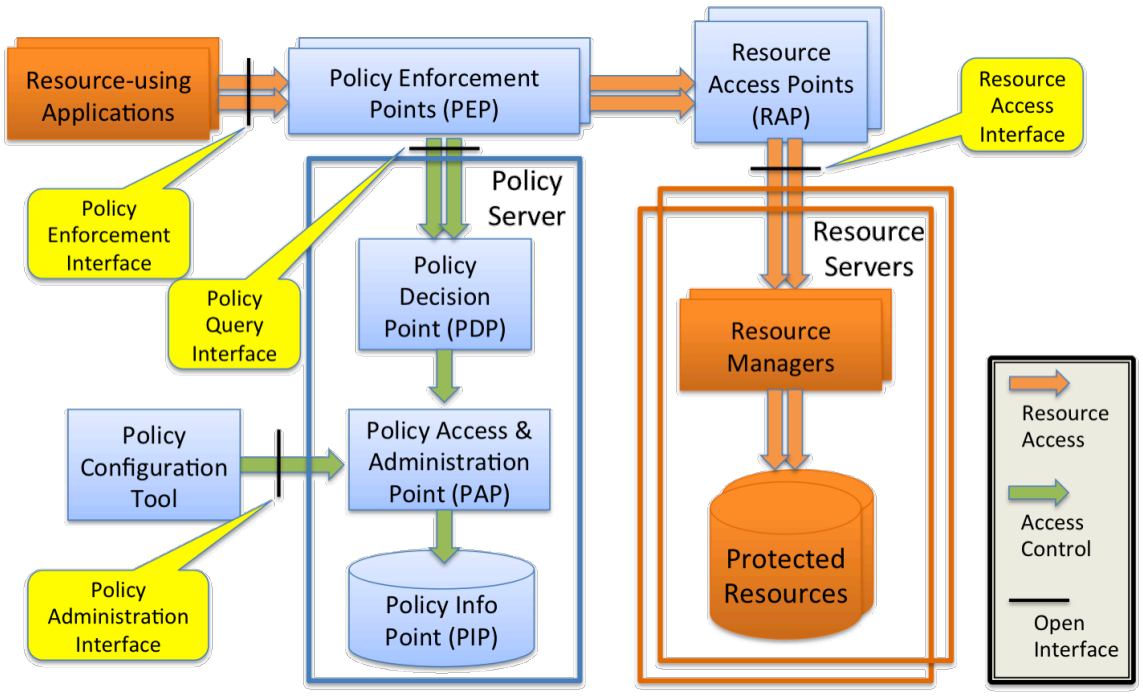


Figure 2: Our NGAC functional architecture with "unbundled" PEP & RAP

In The Open Group's approach the developer creates appropriate PEP and RAP components for new object classes or access methods following simple templates, resulting in separate small and trusted[2] PEP and RAP components. The PEP template includes calls to the Policy Server through the RESTful Policy Query Interface. Besides marshalling and de-marshalling communication parameters, the PEP consists of a single decision based on calling the Policy Server (PDP) and either returning an error condition to its caller if the PDP returned **deny**, or calling the RAP and completing the access operation if the PDP returned **permit** (or **grant**).

The Open Group has pursued design and implementation our own NGAC-related tools and a straightforward declarative language to express policies that comply with the NGAC framework. Specifically, a desktop command-line policy tool called 'ngac' loads policies expressed in the declarative language and can answer queries such as "access(policy1,(u1,r,o2))", the meaning of which is: "under policy 'policy1', is user 'u1' allowed to read object 'o2'?".

As part of the design effort of the PHANTOM project we investigated the feasibility of implementing a portable server to include the PDP/PAP/PIP functions. Through design and experimental implementation we concluded

[2] What makes the components *trusted* is the role they play in the architecture; what makes them *trustworthy* is a combination of their isolation as separate components, their distinct keys for trusted channel communication, their simplicity, and their construction from approved well-developed templates.

that a full heavyweight implementation of a portable server based on a database management system for the PIP, as was the case for a reference implementation of the standard, would be too costly for the project. However, we did discover that we could develop a functional lightweight and portable implementation, the design of which is described in the following.

We expanded our implementation of our first simple policy tool and designed and implemented a server with RESTful APIs for the PEP-to-PDP interface, which we call the Policy Query Interface. The PEP only needs to call the PDP through this interface with an access query that the PDP answers with “permit” or “deny”. Based on this response the PEP must not perform the access to the RAP (if “deny”), or proceed to perform the object access and return the result to the application. The PEP is fundamentally a trivial decision statement conditioned by the PDP’s response that performs the access on one path, or reports an error on the other path.

3. ENHANCED DECLARATIVE POLICY SPECIFICATION LANGUAGE

The declarative language representation is easily constructed from a graphical representation of a policy. The present declarative language does not support the entire NGAC policy framework (lacking prohibitions and obligations) though it is our ambition to add them incrementally in the future as need may require.

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>)

opset(<operation set identifier>, <operations>)

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.

4. 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 policy tool provides the ability to work with a policy as it is being created or modified to test its interpretation by the access calculating algorithms.

4.1 POLICY TOOL INTERACTIVE COMMANDS

The 'ngac' Policy Tool is a command driven application. After starting 'ngac' it offers the prompt "ngac>". There are a set of basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Entering the command "help" will list the available commands in the current mode. Only the most commonly needed commands are introduced here.

- `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.

- `display_policy`.
  Display the specification of the current policy.

- `display_policy(` <policy name> `).`
  Display the specification of the named policy.

- `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.

4.2 COMMAND PROCEDURES AND SCRIPTS

There are predefined 'ngac'command procedures ("procs") that run the examples and can be used for testing and demonstrations. At the "ngac>" prompt a predefined procedure (e.g. named "myproc") can be run with the command `proc(myproc).` It can be run with verbose output with the command `proc(myproc,verbose).` It can be made to prompt and wait for user instruction to proceed (empty line input) with the command `proc(myproc,step).`

It is instructive to read the file procs.pl that defines the predefined procedures. The procedures utilise the same commands available at the command prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

A sequence of 'ngac' commands can also be stored in a file, in which case it is referred to as a *script*. Scripts may be run with a `script` command, analogous to the `proc` command, with the script file name substituted for the stored procedure name. `verbose` and `step` are valid options also for the execution of scripts.

4.3 'NGAC' POLICY TOOL IMPLEMENTATION

The implementation of the 'ngac' policy tool is comprised of the following Prolog modules:

- ngac.pl – top level module of 'ngac' policy tool; entry point and initialisation
- param.pl – global parameters
- command.pl – command interpreter and definition of the 'ngac' commands
- common.pl – simple predicates that may be used anywhere
- pio.pl – input / output of various policy representations
- policies.pl – example policies used for built-in self-test
- test.pl – testing framework for self-test and regression tests
- procs.pl – stored built-in 'ngac' command procedures
- pmcmd.pl – PM RI command representations and conversions
- spld.pl – security policy language definitions

- n_audit.pl – auditing

5. ENHANCED 'NGAC-SERVER' LIGHTWEIGHT 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.

The 'ngac-server' currently provides two external interfaces, both implemented as RESTful APIs:

- Policy Query Interface – used by a Policy Enforcement Point to query whether a given access should be permitted under the current policy.

- Policy Administration Interface – used by a privileged "shell" or "portal" system program to load and unload policies, combine policies, select policies, etc.

Each of these interfaces will now be described in further detail.

5.1 POLICY QUERY INTERFACE (PQI)

A relatively simple interface, in the form of RESTful APIs, constitutes the Policy Query Interface.

This interface is used by a Policy Enforcement Point to determine whether a client-requested operation is supported by the associated user's permissions on the requested object under a particular policy, and if the operation is permitted where may the object be accessed through an appropriate Resource Access Point (RAP).

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

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

5.2 POLICY ADMINISTRATION INTERFACE (PAI)

The Policy Administration Interface is now a separate interface. Policy administration may still be done through the policy tool's command line interface, but it is best done through the server's RESTful Policy Administration API.

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 a ***token*** parameter[3] that acts as a key to use the interface. See the later discussion about protection of the PAI.

Policy Information/Manipulation

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

Parameters

- token = <admin token>

Returns

- <policy identifier> or "failure"

Effects

- none

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

Parameters

- policy = <policy identifier>
- token = <admin token>

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

[3] The default value of the admin token, set in the param.pl file, is 'admin_token'.

- distinguished policy names have special effects: “all” – the composition of all subsequently loaded policies to be applied; “grant” – all queries to return “grant”; “deny” – all queries to return “deny”

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.[4]
- token = <admin token>

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.
- token = <admin token>

Returns

- “success” or “failure”

Effects

- The specified policy element is deleted from the named policy

paapi/combinepol – combine policies to form new policy

Parameters

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

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>

---

[4] 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.

- token = <admin token>

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/loadi – load immediate policy spec into the server

Parameters
- policyspec = <policy specification>
- token = <admin token>

Returns
- "success" or "failure"

Effects
- stores the specified 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>
- token = <admin token>

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

paapi/readpol – read server policy

Parameters
- policy = <policy identifier> (optional, current is default)
- token = <admin token>

Returns
- <policy specification> of named (or current) policy, or "failure"
- "unknown policy" if the named policy is not known to the server

Effects
- no internal effects on server

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>
- token = <admin token>

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>
- token = <admin token>

Returns

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

Effects

- the identified session is deleted from the server

5.3 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`)
- `--deny`, `-d` respond to all access requests with `deny`, sets the current policy to `deny`, which may subsequently be changed by a `setpol` Policy Administration API call
- `--grant, --permit`, `-g` respond to all access requests with `grant`, sets the current policy to `grant`, which may subsequently be changed by a `setpol` Policy Administration API call
- `--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

5.4 DYNAMIC POLICY CHANGE

The Policy Server supports *dynamic total policy change*: the ability to load new policies, to form new policies composed of already loaded policies, and to select from among the loaded or composed policies that policy which is to serve as the policy used to make policy decisions. A policy selection remains in effect until a subsequent policy selection. The server retains all of the loaded and composed policies for the duration of its execution. In addition, the current implementation of the 'ngac-server' offers *limited dynamic selective policy change* after a policy is loaded or formed by combining

policies. The ***add*** and ***delete*** APIs provide this capability. Details of the limitations are provided in the description of the APIs.

The current implementation of the PIP is ephemeral. There is no persistence of the policy database except in the original policy file(s) used to initialize the server and the sequence of commands issued to the server to modify policies after loading of policy files. To recreate a modified policy in a new server instance the original policy must be loaded followed by the same sequence of modifications issued to the server.

5.5 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.

5.6 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.

5.7 AUDITING

The 'ngac-server' generates an audit log of audit records based on the current audit configuration. Auditing may be turned off or audit records may be sent to a log file of the standard error stream, based on the setting of the audit_logging parameter ('off', 'file' or 'on' respectively). Generated audit records are based on the current audit_selection parameter, which may be set to a subset of the auditable_events list enumerated in the file audit.pl. Currently, this setting is done automatically during initialization of the audit module.

5.8 'NGAC-SERVER' POLICY SERVER IMPLEMENTATION

The implementation of the 'ngac-server' lightweight server is comprised of the following Prolog modules:

- server.pl – HTTP server initiation.
- pqapi.pl – the policy query API. (not in v0.3.4+)
- paapi.pl – the policy admin API. (not in v0.3.4+)
- sessions.pl – registration of session identifiers and associated users to enable sessions identifiers to be used in place of the user in an ***access*** request for the life of the session.

- audit.pl – the ‘ngac’ audit module. (n_audit in v0.3.4+)

6. POLICY ENFORCEMENT POINT (PEP) AND RESOURCE ACCESS POINT (RAP) TEMPLATES

The template illustrates the construction of a PEP that queries the PDP and uses a RAP.

The PEP template exhibits the definition of a server for a RESTful Policy Enforcement Interface consisting of the APIs:

- peapi/getLastError – get the error code corresponding to the last error in the API
- peapi/getObject – get an entire object (including opening/closing)
- peapi/putObject – put an entire object (including opening/closing)
- peapi/openObject – open an object for incremental reading/writing
- peapi/readObject – read from an open object
- peapi/writeObject – write to an open object
- peapi/closeObject – close an open object

The RAP template should illustrate how a PEP may access a resource server through a RAP. The example included is a RAP for ordinary OS files, and illustrates file_open, file_close and file_read APIs.

7. INSTALLATION AND OPERATION

7.1 INTRODUCTION

The ‘ngac’ system is composed of two components, the ‘ngac’ Policy Tool and the ‘ngac-server’ Policy Server. The Policy Tool is used to test NGAC policies during their development. The Policy Server uses those policies to provide a runtime policy decision making service. These components share a common set of Prolog source modules and their separate executables are constructed with a simple shell script, ***mkngac***, which accompanies the sources. A primary objective of the ‘ngac’/’ngac-server’ development is to create a lightweight and highly portable access control framework that can be easily adapted to different situations and applications, that has a minimum of external dependencies, and that requires minimal resources to run. This objective has been achieved to a high degree in the current implementation.

7.2 PREREQUISITES

The ‘ngac’ Policy Tool and the ‘ngac-server’ Policy Server are implemented in the Prolog language and require the SWI-Prolog environment to run. This is the software’s sole external dependency apart from the operating system. The version of SWI-Prolog used for the current version is version 7.6.4 available from www.swi-prolog.org.

SWI-Prolog is available for several operating environments, including Microsoft Windows (64 bit) and (32 bit), MacOS X 10.6 and later on Intel, and several Linux versions including Ubuntu. It is also available in Docker containers and as a source distribution that one can build locally.

Our NGAC implementation uses *only* the libraries that come with the Prolog distribution. Furthermore, the functional architecture has been organized so as to place adaptation into the hands of application developers without requiring modification to the core implementation. This is in contrast to the reference implementations that have so many external dependencies (some obsolete) so as to be very cumbersome to work with and not very portable or adaptable.

The software is provided as a set of Prolog source files and/or as “executable” files that have the Prolog runtime environment already linked in. Prior to installing and building the software it is required that Prolog be installed. The exception to this requirement is if there is already a compiled version of ‘ngac’ and ‘ngac-server’ for the target platform, in which case all the dependencies are already linked into the executable.

7.3 INSTALLING AND RUNNING THE ‘NGAC’ POLICY TOOL

The ngac policy tool is implemented in Prolog and requires the SWI Prolog environment to run. The ngac tool can be provided as a set of Prolog source files and/or as an “executable” that has the Prolog runtime environment already bundled in. This executable is made by the shell script mkngac,

located with the source files that must be run in an environment that has SWI Prolog installed.

7.3.1 Install SWI-Prolog

SWI Prolog is available for several operating environments, including Mac, Windows, and Linux. See http://www.swi-prolog.org.

7.3.2 Install the 'ngac' source files and/or executable

The current version of the ngac tool consists of a directory tree including source files and example files. The distribution is provided as a zip file of this directory tree.

7.3.3 Initiate the 'ngac' policy tool

If a ready made executable 'ngac' has been provided it may be executed directly from a command shell prompt. If you do this skip down to "Now you should see …" below.

Otherwise, in the source directory ngac-server-2018-06 start SWI-Prolog from a command shell prompt using the name of the SWI-Prolog executable (usually 'swipl', 'swi-pl', or something similar, depending on how it was installed).

After printing a short banner SWI-Prolog will display its prompt "?- ".

At the Prolog prompt enter "[ngac]." (not the quotes)

Prolog will compile the code and print "true."

Execute the code by entering at the Prolog prompt "ngac."

Now you should see the 'ngac' prompt "ngac> '

7.3.4 Test the installed 'ngac' tool

The ngac tool has some self-tests built in. These should be run to ensure that everything is working correctly. Follow the instructions in the preceding section to run the ngac tool. Start it with the Prolog prompt command "ngac(self_test)." This will run some built-in self tests when it starts. To not run the self-tests simply start the tool with the Prolog prompt command "ngac."

The self tests can also be run by starting 'ngac' normally and entering at the 'ngac' prompt "selftest."

Procedures make up of 'ngac' commands may be predefined in the procs.pl file. Look at the ones there and try them by entering the ngac command

"proc(ProcName).", where ProcName is the name of one of the procedures defined in procs.pl.

7.3.5 Running the examples

There are several examples included with the sources of the ngac Policy Tool. These include examples described in documents and PowerPoint slide decks used to introduce the NGAC concepts.

There are predefined procedures ("procs") that run the examples. At the ngac> prompt a predefined procedure (e.g. named "myproc") can be run with the command `proc(myproc).` It can be run with verbose output with the command `proc(myproc,verbose).`

It is instructive to read the file procs.pl that defines the predefined procedures. The procedures consist of the same commands available at the command prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

7.4 INSTALLING AND RUNNING THE 'NGAC-SERVER'

The ngac server is implemented in Prolog and requires the SWI-Prolog environment to run. The server can be provided as a set of Prolog source files and/or as an "executable" that has the Prolog runtime environment already bundled in. This executable is made by the shell script mkngac, located with the source files that must be run in an environment that has SWI Prolog installed.

7.4.1 Install SWI-Prolog

SWI Prolog is available for several operating environments, including Mac, Windows, and Linux. See http://www.swi-prolog.org.

7.4.2 Install the 'ngac' server source files and/or executable

The current version of the ngac server consists of a directory tree including source files and example files. The distribution is provided as a zip file of this directory tree.

7.4.3 Initiating the 'ngac-server'

The ngac server may started form the 'ngac' policy tool. In normal use the ngac server should be started with a compiled executable. This is preferable since it allows the command line options to be specified.

If you do want to start the server from the policy tool follow the instructions above to get 'ngac' running. After starting 'ngac' it offers the prompt "ngac>". There are a set of basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Entering the command "help" will list the

available commands in the current mode. If you want to load any policy files, do it now with the 'ngac' command "import(policy(PolicyFileName)).", where PolicyFileName is the name of a .pl file relative to the execution directory. You can also combine policies with the 'ngac' "compose" command. When you have the desired policies loaded and composed, start the server from the 'ngac' tool using the command "server(PortNumber).", where PortNumber is an unused TCP port. The server will be started and will be listening to that port for calls to its RESTful API. A server started in this way will expect the default admin token (the string "admin_token", without the quotes) in the policy administration API calls.

The 'ngac-server' can be run in the background from a startup script. To the command 'ngac-server' add any desired command line options and append a "&" to run the process in the background.

7.4.4 Test the installed 'ngac-server'

There is a shell script of curl commands included with the source (servercurltest.sh) in the TEST subdirectory. This script can be run to send a sequence of requests to the server to test for known correct answers. There is also a shell script of curl commands for testing policy composition (serverCombinedtest.sh). (Note: These scripts run illustrative tests, they do not run a complete set of tests.)

7.5 CONFIGURATION

The tool can be easily extended in several ways.

- Commands can be added by modifying the `command` module to add a `syntax`, `semantics` (optional), `help`, and `do` clause for the new command. A `syntax` clause must be added for the command. This clause declares the command name and parameters, and what mode the command belongs to, admin or advanced. Admin commands are available in admin mode, but also accessible in the advanced mode but not vice versa.

- The self-test framework is implemented in the `test` module. Tests for specific new modules can be added in the `TEST` subdirectory. An example of a test definition file for the `spld` module is implemented in `TEST/spld_test.pl`.

- New predefined 'ngac' command procedures can be added to the `procs` module. A `proc` clause is added for each new procedure to be defined. There are examples in the `procs.pl` file.

Global parameters are set in the `param` module. Settable parameters (those that can be changed from the 'ngac' command line with the `set` command)

are itemized in a list `settable_params`. For example, the parameter audit_logging is a settable parameter. Adding new settable parameters requires the new parameter name to be added to the list of settable parameters and to the `dynamic` directive above it in a fashion similar to the other entries.

8. INTEGRATING NGAC WITH AN EXISTING SYSTEM

8.1 ADAPTING TO THE NGAC FUNCTIONAL ARCHITECTURE

Generally, when you take direct resource access code out of the application and replace it with PEP interface references the removed code will be represented in some form in the RAP. The PEP should be limited to marshaling the arguments to the PDP access call and determining what RAP to invoke and the needed parameters. Depending on the narrowness of the set of resources handled by the PEP (there can be multiple PEPs) the RAP call side of the PEP could be fairly simple. In fact, it is acceptable to combine the PEP and the RAP into a single execution unit (while keeping the functions separate) if the association of the PEP and RAP are 1-1. Since the RAP is now be acting for potentially multiple resource access references in one or more applications, it will be more general than any one individual reference. If there are multiple resource "locations" serviced (local or remote, for example) then it may be best to keep the PEP simpler by passing the location to the RAP and having the RAP access the proper location.

8.2 DEPLOYING THE NGAC COMPONENTS

The ngac server does not need to be running in any one particular place, but it should be used by all PEP/RAPs. It is not a strict rule but, generally, having the PEP close to the client app and having the RAP close to the resource makes sense, unless the PEP and RAP are combined, in which case a decision must be made where to deploy it.

8.3 CREATING A POLICY

1) identify the distinct objects to be protected (protected resources)
2) identify the set of possible operations on each object
3) identify the distinct users using identifiers that can be determined at runtime
4) for the users and for the objects determine a set of (binary) attributes
that are relevant to making policy decisions (that is attributes that a user
or an object either has, or does not have). Often it is convenient to create
attributes that make sense for the domain whether or not they correspond
to actual runtime entities.
5) make the appropriate assignments of users to user attributes,
user attributes to other user attributes, objects to object attributes, and
object attributes to other attributes.
6) make the graph connected by having the user side and the object side
both belong to the same policy class, and by having the policy class
belong to the connector 'PM' as in the diagram for the example policy.

Now, for web services I suggest to adapt the methodology slightly.
For this case I'm going to reuse some ideas from the example attached.
One should still identify the distinct objects (or resources) and the operations
permitted on them. Often there are multiple operations on the same object.
So now you must make a decision, for the policy you are going to create,
whether to have each Web API (URI) be a distinct *object*, or whether to

have the Web APIs be distinct *operations* on an underlying object.

Suppose you provided the contents of a file as a web service. There are multiple ways to package this. One way is to provide a separate read and write operations on the file object

fileAservice read
fileAservice write

which could also look like the APIs

fileAserviceRead
fileAserviceWrite

In the policy this could be:
policy(_, _. [
object(fileA),
object_attribute(served_file),
assign(fileA,served_file),
associate(ordinary_user,[read],served_file),
…]).

queries would look like access <u1,read,fileA>
where the operations are read and write

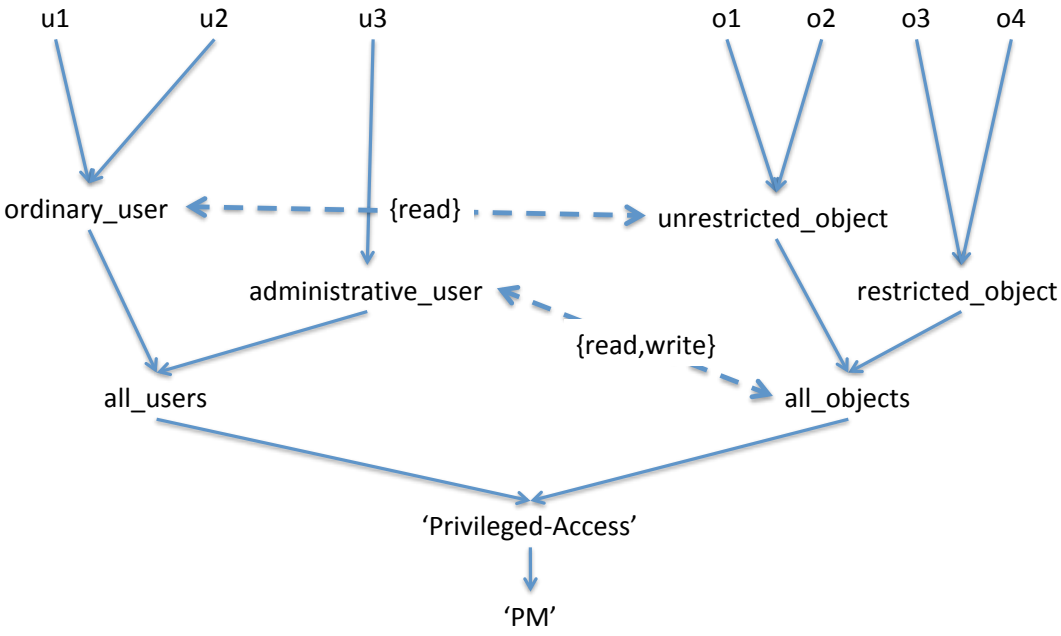
or:
policy(_, _. [
object(fileAread),
object(fileAwrite),
assign(fileAread,unrestricted_API),
assign(fileAwrite,restricted_API),
associate(ordinary_user,[call],unrestricted_API),
associate(administrative_user,[call],all_APIs),
…]).

queries would look like access <u1,call,fileAread>
where the operation is call

it all depends on how you want to organize the concepts in the policy and whether you want to think of every API (URI) as an independent resource (even if it operates on the same underlying object) or think of potentially multiple APIs providing different operations on the same underlying object.

The Policy Enforcement Point for a web service could be a proxy for the service that calls the PDP. It is essential that a user of the service can only access the service through the PEP proxy.

Example ‘Policy4’: ‘Privileged-Access’



Example ‘Policy4’: ‘Privileged-Access’

```
policy('Policy4','Privileged-Access', [

    user(u1),
    user(u2),
    user(u3),

    user_attribute(ordinary_user),
    user_attribute(administrative_user),
    user_attribute(all_users),

    object(o1),
    object(o2),
    object(o3),
    object(o4),

    object_attribute(unrestricted_object),
    object_attribute(restricted_object),
    object_attribute(all_objects),

    policy_class('Privileged-Access'),

    connector('PM'),

    assign(u1,ordinary_user),
    assign(u2,ordinary_user),
    assign(u3,administrative_user),

    assign(ordinary_user,all_users),
    assign(administrative_user,all_users),

    assign(o1,unrestricted_object),
    assign(o2,unrestricted_object),
    assign(o3,restricted_object),
    assign(o4,restricted_object),

    assign(unrestricted_object,all_objects),
    assign(restricted_object,all_objects),

    assign(all_users,'Privileged-Access'),
    assign(all_objects,'Privileged-Access'),

    assign('Privileged-Access','PM'),

    associate(ordinary_user,[read],unrestricted_object),
    associate(administrative_user,[read,write],all_objects)
]).
```


8.4 ENFORCING THE NGAC FUNCTIONAL ARCHITECTURE

The components of the NGAC functional architecture can only do their intended functions, and the architecture achieve its intended benefits, in the face of a hostile environment, if they can operate without malicious interference. That is, the functional architecture must be affirmatively *enforced*. Since the NGAC components run with the existing system as their “IT environment” it is necessary to embed the NGAC components within the environment in a way that achieves the two essential properties of a reference validation mechanism (aside from the obvious first one: correctness), that is, tamper-proof-ness and non-bypassability (“always invoked”). These properties cannot be achieved by the mechanism itself, but must be provided for the mechanism by its environment through a proper embedding and use of the native protection features of the environment. To accomplish this, particularly in the case of distributed systems with many kinds of protected resources, may not be a trivial matter.

We note that some deployments of NGAC are done with the intention of demonstrating the utility or benefits of a common attribute-based access control system such as NGAC within a particular application. In the case of such benign environments, it is the proof-of-concept of the utility of a unified access control system that is the goal, not absolute and complete robustness of the deployment in the demonstrator. As long as it is feasible, in principle,

to achieve the enforcement of the functional architecture, we argue that it may not be justified to expend the resources to achieve that enforcement in the deployment. We have found this to be the case in some of our projects in which the goal is to demonstrate the utility of fine-grained access control in new contexts where it can address a resource protection challenge that is unique to, or exacerbated by, the application concerned.

For the purpose of this discussion on the deployment of NGAC we assume the environment to be a general-purpose operating system or embedded operating system that provides basic protections, such as process integrity, process identity, file object integrity, and file access controls. For the sake of example we shall assume a Unix-like operating system or one providing similar features in the area of basic protections mentioned above.

Let us begin by outlining the requirements:

Specifically, and at a minimum, the PEP, RAP, and PDP/PAP/PIP should run as distinct processes that should be run with a reserved user identity (we'll call it user *ngac*).

The Policy Query Interface of the PDP and the Resource Access Interface of the RAP should be restricted to be callable only by PEPs.

The resources to be controlled by NGAC should be made to be accessible *exclusively* to the ngac user or otherwise limited to access only by the RAP through the corresponding resource server.