Project Number 723634

D5.4 Full Prototype of the SPT Framework

Version 2.2
28 June 2020
Final

Public Distribution

The Open Group

Project Partners: **ATB, Electrolux, IKERLAN, OAS, ONA, The Open Group, University of York**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the SAFIRE Project Partners accept no liability for any error or omission in the same.

© 2020 Copyright in this document remains vested in the SAFIRE Project Partners.

PROJECT PARTNER CONTACT INFORMATION

ATB Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de	**Electrolux Italia** Claudio Cenedese Corso Lino Zanussi 30 33080 Porcia Italy Tel: +39 0434 394907 E-mail: claudio.cenedese@electrolux.it
IKERLAN Trujillo Salvador P Jose Maria Arizmendiarrieta 20500 Mondragon Spain Tel: +34 943 712 400 E-mail: strujillo@ikerlan.es	**OAS** Karl Krone Caroline Herschel Strasse 1 28359 Bremen Germany Tel: +49 421 2206 0 E-mail: kkrone@oas.de
ONA Electroerosión Jose M. Ramos Eguzkitza, 1. Apdo 64 48200 Durango Spain Tel: +34 94 620 08 00 jramos@onaedm.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of York Leandro Soares Indrusiak Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325 570 E-mail: leandro.indrusiak@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial structure and outline	1 December 2018
0.5	Initial content	23 December 2018
1.0	Final version	31 December 2018
2.0	First content updates addressing review recommendations	21 April 2020
2.1	Additional updated content	29 May 2020
2.2	Final content addressing review recommendations	28 June 2020

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

Abbreviations and Acronyms

AD	Active Directory
AP	Abstract Platform
API	Application Programming Interface
App	Application
EP	Early Prototype
EPP	Event Processing Point
FoF	Factories of the Future
FP	Final Prototype
HW	Hardware
I&A	Identification and Authentication
IoT	Internet of Things
IIoT	Industrial Internet of Things
IIC	Industrial Internet Consortium
IIRA	Industrial Internet Reference Architecture
IISF	Industrial Internet Security Framework
Impl	Implemented
IT	Information Technology
LDAP	Lightweight Directory Access Protocol
NGAC	Next Generation Access Control
OE	Operating Environment (operating system)
OS	Operating System (operating environment)
PAP	Policy Access/Administration Point
PDP	Policy Decision Point
PEI	Policy Enforcement Interface
PEP	Policy Enforcement Point
PImpl	Partially Implemented
PIP	Policy Information Point
PM	Policy Machine
PQI	Policy Query Interface
RAI	Resource Access Interface
RAP	Resource Access Point
RI	Reference Implementation
SOPS	System of Production Systems
SPTM	Security Privacy and Trust Methodology/Mapping
SSF	SAFIRE Security Framework (or SAFIRE SPT Framework)
SW	Software
TBI	To Be Implemented
UC	Use Case
3PSM	Third Party Security Mechanisms

EXECUTIVE SUMMARY

This document describes the full prototype of the software implementing the functionality of the SAFIRE Security Privacy and Trust Framework as specified in deliverable D5.5 and implementing the methodology described in deliverable D5.1, thereby demonstrating the features and functionality of security services for the SAFIRE solution.

The document provides a description of the SAFIRE implementation of the NGAC standard, including the policy tool, the policy server, and the policy specification language. It also describes the integration with other modules and the functional architecture within which the components are to be deployed, and examples of how they can be used. Installation and Operation of the full prototype software are addressed, along with guidance for customisations for addressing diverse manufacturing applications where SAFIRE may be deployed.

1. INTRODUCTION

1.1. OVERVIEW

As introduced in D5.2 Early Specification of the SPT Framework, and finalized in D5.5 Final Specification of the SPT Framework, the SAFIRE security, privacy and trust effort intends to address two SAFIRE-relevant security challenges, and has two principal thrusts in SAFIRE. These comprise the SAFIRE Security Framework (or SAFIRE SPT Framework) (SSF):

- To address Security Challenge 1, that of being able to know the aggregate security policy of the various policies independently enforced, according to their configuration data, by distinct security mechanisms used within a complex system: An implementation of the Next Generation Access Control (NGAC) standard, that will provide the ability to specify aspects of policies corresponding to distinct segments or mechanisms within the system, and to compose those policy aspects to gain a unified and computable representation of the composed policy.

- To address Security Challenge 2, that of being able to assess the initial adequacy of the constellation of chosen security mechanisms and their respective configurations, and their continuing adequacy after a reconfiguration or optimisation has been applied: A comprehensive approach to security using the Industrial Internet Consortium (IIC) Security Framework (IISF) as guidance for the kinds of security functions that should be considered in a mission-critical industrial deployment of SAFIRE within a System of Production Systems (SOPS) Factories of the Future (FoF) setting.

1.2. APPROACH APPLIED

The first thrust, identified above, is realized by a software implementation of the NGAC standard that embodies one of the innovations of the SAFIRE project. The second thrust is realized by a methodology to assess the appropriateness and adequacy of security mechanisms chosen for a particular deployment, including those of the underlying platform as well as those provided by the NGAC implementation.

This document, D5.3 Early Prototype, describes the content of the first prototype of the NGAC implementation. The companion document, D5.1 Methodology, elaborates the methodology for the application of the IISF as well as methodology for the use of the software components of the NGAC policy specification and enforcement mechanisms.

The SAFIRE software portion of the SPT Framework includes:

- The policy tool (‘ngac’)
- The lightweight server (‘ngac-server’)
- A functional architecture for using the ‘ngac’ tool and ‘ngac-server’

1.3. Document Structure

The document consists of:

- Section 1. Introduction—which describes the purpose of this document, and provided a brief overview of the contents of the document.
- Section 2. Description of the NGAC Full Prototype (FP)—our implementation of the NGAC standard, including the policy tool, the policy server, and the policy specification language.
- Section 3. Integration with other modules—describes the functional architecture in which the NGAC components are to be applied, and examples of how they should be used.
- Section 4. Installation and Operation—describes how to install and operate the NGAC FP components and how to invoke them from other components of the functional architecture.
- Section 5. NGAC Customisation—describes how NGAC can be customised for diverse applications.
- Section 6. Software Tools—identifies the software tools used for the implementation.
- Section 7. Conclusions and Plans.
- Section 8. References.
- APPENDIX A—Provides expanded background on NGAC.
- APPENDIX B—Provides a brief discussion of NGAC-based policy specification representations.
- APPENDIX C—Presents a methodology for developing an NGAC policy and expressing it in the declarative policy language, including development of a policy for ONA.
- APPENDIX D—Provides an analysis of opportunities to apply NGAC within the Industrial Internet Security Framework.

2. Description of the NGAC Full Prototype (FP)

The functional architecture of NGAC as realized in the SAFIRE Project is depicted in Figure 1. We will refer frequently to the components of the functional architecture introduced in this figure. A summary of NGAC can be found in APPENDIX A – Next Generation Access Control, and standards and other background can be found in documents enumerated in Section 11 REFERENCES.

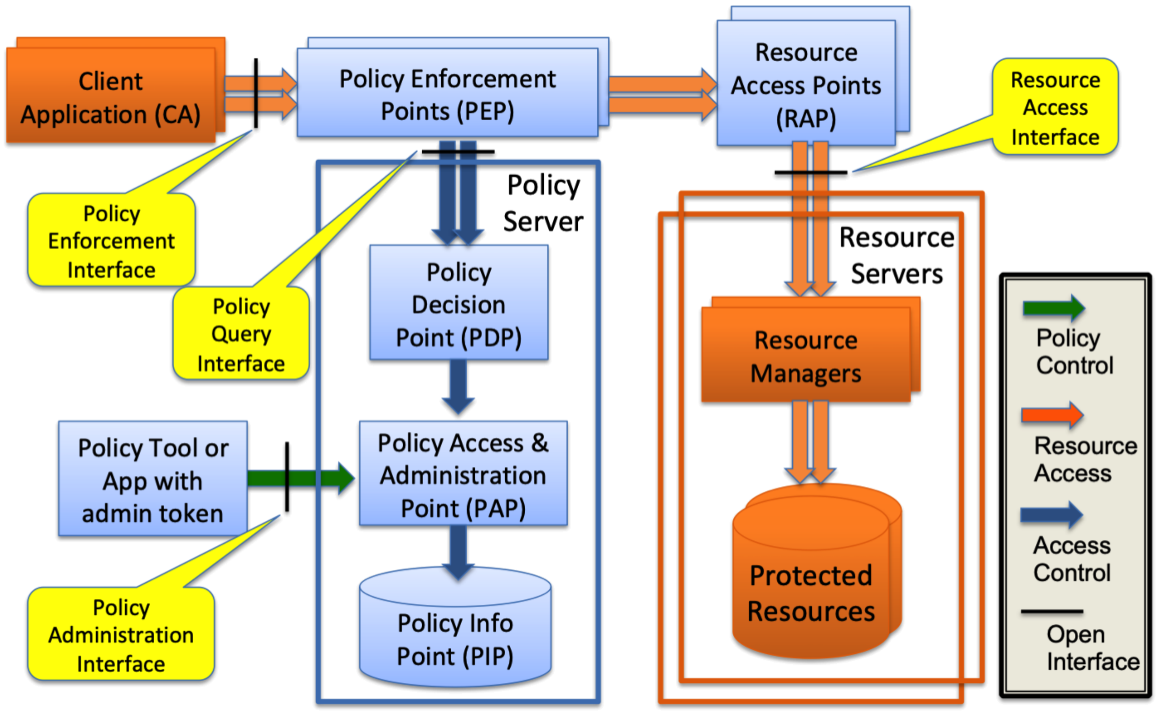

Client Application (CA)
Policy Enforcement Points (PEP)
Resource Access Points (RAP)
Resource Access Interface
Policy Enforcement Interface
Policy Query Interface
Policy Server
Policy Decision Point (PDP)
Resource Servers
Resource Managers
Policy Tool or App with admin token
Policy Access & Administration Point (PAP)
Protected Resources
Policy Administration Interface
Policy Info Point (PIP)
Policy Control
Resource Access
Access Control
Open Interface

Figure 1: TOG NGAC functional architecture with "unbundled" PEP & RAP

The logical and functional components of the NGAC Full Prototype implementation for SAFIRE are:

- Enhanced declarative policy specification language (DPL) supporting modular policies and policy composition;
- Enhanced Policy Tool (‘ngac’) for doing standalone policy development and testing; recognizes and provides semantics for the DPL, providing the ability to query a policy under development;
- Policy Server (‘ngac-server’) with RESTful APIs providing the Policy Query Interface and the Policy Administration Interface;
    - Policy Access Point (PAP) implementing the Policy Administration Interface and a Policy Information Point (PIP) policy store, also provided within the Policy Server;

- Policy Enforcement Point template providing the Policy Enforcement Interface; and,
- Resource Access Point template using the Resource Access Interface.

Each of these components will be described in the following sections.

2.1. Declarative Policy Language

The NGAC full prototype represents a lightweight implementation of the NGAC standard. During the course of the implementation of NGAC we have developed alternative policy representations, including an alternative representation of the reference implementation's imperative language and our own declarative policy language (DPL). We envision future development of the DPL to include additional features to facilitate the specification of complex policies for enterprise-wide use and conditional dynamic policy change.

The declarative policy language recognized by the 'ngac' policy tool and the 'ngac-server' policy server is used to directly declare the entities of a policy and the relations among them. The declarative language is an attractive alternative to using an imperative policy construction language as it is much more natural and intuitive.

2.1.1. Declarative Policy Language Definition

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(<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 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**. (this parameter is currently not used)

< *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.

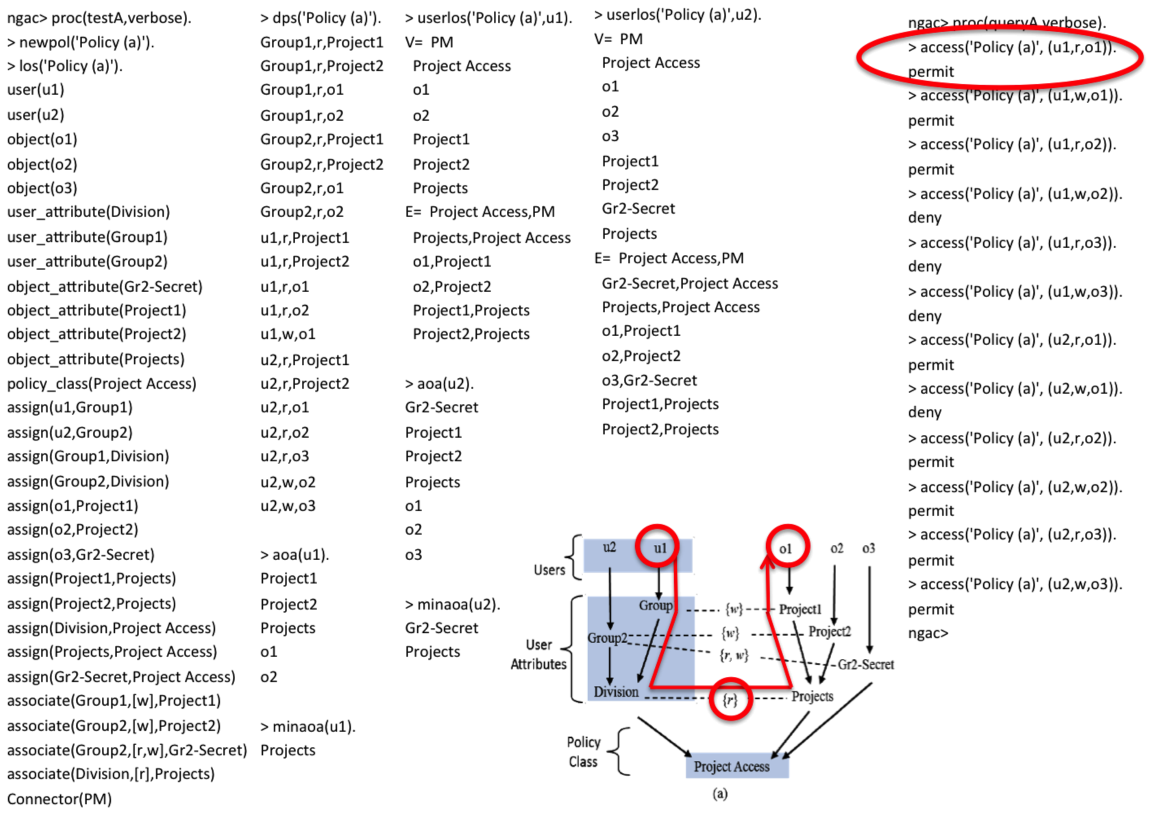
2.1.2. Example policy in declarative representation

In APPENDIX B, Section 13.1, Figure 11, two examples, (a) and (b), of NGAC policies are presented in graphical form. These examples provide a good generic reference because they are simple, well developed and well explained in the NGAC literature. Figure 2 presents example (a) in the declarative policy language representation.

The figure represents a run of the ‘ngac’ policy tool in which the stored procedure “testA” is run with the verbose switch on. The procedure sets the current policy to ‘Policy (a)’ and runs some analysis queries against the policy, including, in the right-hand column, access queries such as:

`access(‘Policy (a)’, (u1,r,o1)),`

highlighted in Figure 2 with a red ellipse. Inset in the figure is the graphical representation with the path in the graph highlighted that enables the **`permit`** response.



```
ngac> proc(testA,verbose).
> newpol('Policy (a)').
> los('Policy (a)').
user(u1)
user(u2)
object(o1)
object(o2)
object(o3)
user_attribute(Division)
user_attribute(Group1)
user_attribute(Group2)
object_attribute(Gr2-Secret)
object_attribute(Project1)
object_attribute(Project2)
object_attribute(Projects)
policy_class(Project Access)
assign(u1,Group1)
assign(u2,Group2)
assign(Group1,Division)
assign(Group2,Division)
assign(o1,Project1)
assign(o2,Project2)
assign(o3,Gr2-Secret)
assign(Project1,Projects)
assign(Project2,Projects)
assign(Division,Project Access)
assign(Projects,Project Access)
assign(Gr2-Secret,Project Access)
associate(Group1,[w],Project1)
associate(Group2,[w],Project2)
associate(Group2,[r,w],Gr2-Secret)
associate(Division,[r],Projects)
Connector(PM)
```

```
> dps('Policy (a)').
Group1,r,Project1
Group1,r,Project2
Group1,r,o1
Group1,r,o2
Group2,r,Project1
Group2,r,Project2
Group2,r,o1
Group2,r,o2
u1,r,Project1
u1,r,Project2
u1,r,o1
u1,r,o2
u1,w,o1
u2,r,Project1
u2,r,Project2
u2,r,o1
u2,r,o2
u2,r,o3
u2,w,o2
u2,w,o3

> aoa(u1).
Project1
Project2
Projects
o1
o2

> minaoa(u1).
Projects
```

```
> userlos('Policy (a)',u1).
V= PM
  Project Access
  o1
  o2
  Project1
  Project2
  Projects
E= Project Access,PM
  Projects,Project Access
  o1,Project1
  o2,Project2
  Project1,Projects
  Project2,Projects

> aoa(u2).
Gr2-Secret
Project1
Project2
Projects
o1
o2
o3

> minaoa(u2).
Gr2-Secret
Projects
```

```
> userlos('Policy (a)',u2).
V= PM
  Project Access
  o1
  o2
  o3
  Project1
  Project2
  Gr2-Secret
  Projects
E= Project Access,PM
  Gr2-Secret,Project Access
  Projects,Project Access
  o1,Project1
  o2,Project2
  o3,Gr2-Secret
  Project1,Projects
  Project2,Projects
```

```
ngac> proc(queryA,verbose).
> access('Policy (a)', (u1,r,o1)).
permit
> access('Policy (a)', (u1,w,o1)).
permit
> access('Policy (a)', (u1,r,o2)).
permit
> access('Policy (a)', (u1,w,o2)).
deny
> access('Policy (a)', (u1,r,o3)).
deny
> access('Policy (a)', (u1,w,o3)).
deny
> access('Policy (a)', (u2,r,o1)).
permit
> access('Policy (a)', (u2,w,o1)).
deny
> access('Policy (a)', (u2,r,o2)).
permit
> access('Policy (a)', (u2,w,o2)).
permit
> access('Policy (a)', (u2,r,o3)).
permit
> access('Policy (a)', (u2,w,o3)).
permit
ngac>
```

Figure 2: Example policy (a) expressed in the declarative representation

The 'ngac' tool has also been implemented with a built-in test framework that permits a customizable set of test cases to be easily integrated and run as a regression test suite with a single command. The test cases include of both access queries that correspond to the Policy Query Interface of Figure 1 and other testing queries that expose intermediate internal results of policy calculations for diagnostic use as are shown in Figure 2. The 'ngac" policy tool is described in more detail in the following section.

2.1.3. Implementation

Policies in the NGAC policy framework are represented as a directed graph, formed by users, objects, attributes, and relations among these. A policy developer may draw a policy graph diagram and directly translate it into the declarative policy language.

We chose to implement NGAC in Prolog, which is well suited to this problem for several reasons, among which are the natural way such entities and relations can be represented in Prolog, its ease of expressing and compactness of graph computations (in large part declaratively), the ease of doing linguistic and symbolic manipulations, its useful libraries, and its suitability for prototyping. Conversions among various internal and alternative external representations of languages and structures are easily accomplished in Prolog. Little code is needed to convert the external representation of the DPL into an internal form that is more suitable for performing policy computations.

The internal form of one or more policy specifications expressed in the DPL are stored in the PIP through the PAP. From there they may be selected, referenced and combined by functions of the PAP, and they may be used by the PDP for making policy decisions. The semantics of a stored policy are provided by the graph computations over the internal form of the policy graph in the PDP.

The external syntax of the DPL is embedded in the general syntax of terms as are amenable for reading by the Prolog reader routines. As used by the DPL, the syntax elements borrowed from Prolog are only the characters "(", ")", "[", "]", ",", " ' ", and ".". All other elements of the policy language are alphanumeric names (including "_") and numbers. Names that start with a capital letter must be placed in single quotes. Names starting with a lower-case letter need not be quoted, though they may be.

2.2. 'NGAC' POLICY TOOL

The 'ngac' policy tool enables standalone policy development and testing of NGAC policies. This can be very useful as it does not require a user to arrange for an instance of the policy server to be run at a particular port, or to be concerned with the additional complication of constructing programs to invoke the policy server's APIs to load and test the policy.

The policy tool also provides its user with the ability to examine at the storage of policies and intermediate results of policy calculations. The tool also has the ability to start an instance of the policy server when the user is ready to test an access policy that is used by a client application that accesses resources through an NGAC Policy Enforcement Point (PEP). During such a session the user may continue to interactively test and modify the policy that the server is using to respond to HTTP queries from the client application being tested. This provides flexibility in how testing can progressively be done as a policy is developed and transitioned into production use.

The 'ngac' Policy Tool is an interactive command driven application. After starting 'ngac' it offers the prompt "ngac>". There are basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Advanced commands not accepted in administrator mode. Entering the command "help" will list the available commands in the current mode. The commands `admin` and `advanced` are used to change modes. By default the 'ngac' tool starts in advanced mode; this can be changed in the parameter file and the tool recompiled.

The 'ngac' policy tool uses the declarative policy representation to build an internal database that represents the graph of the policy and associated information, and it includes logic to query the database and to perform graph analysis to compute access decisions. The tool also includes commands to load and manipulate policies.

2.2.1. 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 normal mode (admin) 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.
- `dps(`<policy name>`).`
  Show derived privileges of the specified 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.
- `policy_graph.`
  Display the current policy. Temporary files are created in the GRAPHS directory and removed at the end of the command execution. The GRAPHS directory is created if it does not exist. The rendered image is displayed on the console.
- `policy_graph(`<policy name>`).`

Display the named policy. The specified name can be "current_policy". Temporary files are created in the GRAPHS directory and removed at the end of the command execution. The GRAPHS directory is created if it does not exist. The rendered image is displayed on the console.

- `policy_graph(`<policy name>`,` <file base name> `).`
  Generate the graph for the named policy and store the Dot language version in the file GRAPHS/<file base name>.dot and the rendered graph in the file GRAPHS/<file base name>.png. The GRAPHS directory is created if it does not exist. The rendered image is displayed on the console.

- `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.

- `quit.`
  Terminate the ngac command loop or script, but stay in Prolog top level.

- `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 policy server on the given port number, and require given admin token to be supplied by callers as authenticator.

- `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, additional advanced user commands to support development and diagnostics.

2.2.2. 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.

2.2.3. Policy Graph Display

A graphical rendering of loaded policies may be displayed on the console and optionally sent to a file by the `policy_graph` commands. To generate the graphical display the 'ngac' Policy Tool converts the policy specification into a stylized-for-NGAC-policies graph description in the Dot language, which is subsequently rendered using the dot/graphviz tools.

The graph display capability is new and experimental. An effort has been made to cause the graphs to be laid out in the manner that is customary for NGAC policies, and our examples produce acceptable results. Nonetheless, some complex policies may produce unexpected results; which is expected to improve over time with tuning as the technology matures. The implementation is structured to allow additional layout heuristics to be incrementally added. With additional examples and tuning it is expected that pleasing output can be produced for an increasing number of cases.

Some examples of the `policy_graph` feature are included in this updated version of D5.4 in Appendices B and C.

2.2.4. Implementation

The 'ngac' policy tool is structured as an interactive or script-driven command interpreter that invokes a collection of command functions and supporting functions. The Policy Tool defines the 'ngac' command language syntax, semantics, and help text as a collection of declarations that can be easily extended. The command interpreter's top-level loop reads a user command and then uses the provided definitions to check the command before calling the associated command function to carry out the requested command. The command interpreter also offers developer aids such as internals inspection and selective tracing of commands for development and debugging of the tool itself.

The command functions are grouped according to categories: definition and processing of the declarative policy language, input and output of policy representations, policy representation conversions, computations over the internal policy representation including access queries, an extensible self-test framework with built-in test cases, and a few utility functions. The command interpreter can also run 'ngac' command *procedures* that may be predefined in one of the source files of the tool or in scripts that may be provided in files by the user.

To summarize, the ‘ngac’ command interpreter is thus extensible in several ways:

- The addition of new commands to directly access or combine any functionality already available within the tool, or with a new command function.
- A self-test framework implemented in the test module permits addition of tests for specific modules. These tests can be run automatically at startup or on demand.
- Predefined ‘ngac’ command procedures may be added to the procs module.
- Global parameters are settable in the param module.

The syntax and simple semantic checking of commands are achieved declaratively, and the addition of a new command is straightforward. There are two levels of commands: a restricted set for ordinary policy administrators and an expanded set that includes commands that are primarily of use to the tool developers. The current command set is determined by the current value of a parameter and it may be changed by a command. The extension capability has been used heavily during development and will be useful also as capabilities continue to be added in the future.

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 command representations and conversions
- spld.pl – security policy language definitions

The ‘ngac’ tool is able to generate from the declarative form of a policy the equivalent imperative form for interoperation with the reference implementation. In the future this capability will be adapted to extend the testing framework to support testing of dynamic policy change in the server by generating calls to the policy administration APIs of the policy server to effect incremental changes to policies currently loaded into the server, thereby simulating policy modification by administrative programs using the Policy Administration Interface.

2.3. ‘NGAC-SERVER’ POLICY SERVER

The ‘ngac-server’, developed primarily on the SAFIRE project, is described as “lightweight” as it is much smaller, much easier to extend, virtually no external dependencies and much more portable than the past reference implementations.[1] The Policy Server embodies the following components of the NGAC Functional Architecture: Policy Decision Point (PDP), Policy Access/Administration Point (PAP), and the Policy Information Point (PIP) for policies actively in use at run time.

The policy server may be initiated from within the ‘ngac’ tool by issuing the command `server( <port> ),` or other variations of the command, at the tool’s command prompt “ngac>”. It may also be initiated by executing the compiled file at the operating system’s command prompt.

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”, “portal” or other system program to load and unload policies, combine policies, select policies, etc., for NGAC client applications that will subsequently be run by the shell.

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

2.3.1. Policy Query Interface (PQI)

The PQI is a relatively simple interface, in the form of RESTful APIs.

This interface is used by a Policy Enforcement Point to determine whether a client application-requested operation is supported by the associated user’s permissions on the requested object under the current policy, and, if the operation is permitted and the resource location is not already known to the PEP, where the object may 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

[1] Due to serious difficulties encountered in working with the reference implementations on past

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. Session management APIs are among the functions available in the Policy Administration Interface.

2.3.2. Policy Administration Interface (PAI)

The Policy Administration Interface is now a separate interface (different to the EP). As in the EP, policy administration may still be done through the policy tool's command line interface, but in the FP it is best done through the server's RESTful Policy Administration API.

In keeping with the principle of least privilege, 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, 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. Putting the policy administration APIs in a different location will facilitate limiting their accessibility. Further, the policy administration API requires pre-authentication and the presentation of a token with each API call that the caller obtains through a trusted handoff.

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. Implicit in each of the following APIs is an additional parameter, **token**, which is an arbitrary string. The token provided in the call must match the token provided to the server when it was started.

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

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.

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/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/combinepol – combine policies to form new policy

Parameters

- 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/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

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

2.3.3. Policy server command line arguments

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

- `--token`, `-t` <admintoken> use the token make authenticated requests to the paapi
- `--deny`, `-d` respond to all access requests with `deny`
- `--permit`, `--grant` `-g` respond to all access requests with `grant`
- `--import`, `--policy`, `--load`, `-i`, `-l` <policyfile> import the policy file on startup
- `--port`, `--portnumber`, `--pqport`, `-p` <portnumber> server should listen on the specified port number
- `--selftest`, `-s` run self tests on startup
- `--verbose`, `-v` show all console messages

2.3.4. Protection of 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. The administrative token is used to recognize authorized callers. The system initialization procedure that starts the policy server generates a new administrative token, communicates it to the server though the ***token*** command line option, and distributes it to processes that are authorized to call the PAAPI. The initiation procedure is discussed in more detail in Section 3.6.

If the Policy Server is started without the ***token*** option it will use the default admin token defined in the parameter 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 with a fresh token an untrusted process would be able to manipulate the policy being used by the Server, effectively *tampering* with the access control reference monitor.

2.3.5. Dynamic policy change

However, it does support *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.

2.3.6. 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 server’s 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. 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 must be designed with the foregoing ***access*** runtime semantics taken into consideration. Under normal composition, ***access*** consults all policy classes in the applied policy

2.3.7. Auditing

The ‘ngac-server’ provides a basic auditing facility for use within NGAC that generates a log of audit records based on the current *audit configuration*. The audit facility is accessed through an internal interface consisting of audit_gen, audit_set, audit_select, and audit_deselect operations. The generation of audit events is done by calls to the audit_gen operation, which is used to report auditable events. The information associated with an audit event is: a timestamp of the time of creation of the audit event, the source module or subsystem, the event name, and arbitrary data associated with the event.

A reported audit event results in an audit log record if the event corresponds to one of a set of currently selected audit events established with audit_set, audit_select, and audit_deselect. The audit records for selected events are forwarded to the system logging facility through an OS-provided interface and/or they may be stored in a local audit log file according to a server configuration option. A straightforward extension would enable audit records to be sent over a trusted channel to an audit server in cases where greater integrity of the audit storage is required.

2.3.8. Implementation

The implementation of the ‘ngac-server’ lightweight policy server is comprised of the following Prolog modules:

- server.pl – HTTP server for the policy query API and policy admin API

- 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.

2.4. Requirements, Objectives and results of the FP

2.4.1. Setting the objectives

SAFIRE is conceived as an add-on to an existing manufacturing system. It is intended to enhance the operation of its host system by its presence but should not prevent the host system from operating by its absence. This presents some challenges for the security component of the SAFIRE effort because of the interplay between the existing security features of the host system and of its supporting platform with existing but not necessarily compatible security features of the additional third party ICT technology components used by SAFIRE (such as Kafka, NiFi, Docker, Spring, Cassandra and Spark).

We presented rationale for the approach taken to the SPT Framework in the Final Specification of SPT Framework [SAF D5.5], sections 2 and 3. We also noted the scope of those functional building blocks identified by the IISF that were implemented as part of the SAFIRE effort.

An industrial environment into which SAFIRE is introduced already represents a unique identified *security problem*. The implementation of a solution to the security problem is complex and costly. It is the purview of an information systems department of the organisation that owns and operates the underlying IT infrastructure, and perhaps that a more specialized information systems security unit within the information systems department.

The solution to the security challenges faced by the underlying system depend on the assessed threats faced, the value of the assets, the information security policies of the organisation, the protections afforded by the operational environment, the choices for the underlying information systems technology, the chosen security features, and the security products chosen to provide those features. The assembly of security products, operational procedures, and personnel practices form a comprehensive solution that suits the needs of the organisation.

Because of the magnitude of this first challenge, the second challenge for the SAFIRE Project becomes deciding on what specific efforts to pursue in order to enhance the SAFIRE solution and provide incremental benefit to the underlying system with the necessarily limited resources allocated to SPT within the SAFIRE project.

Thus we sought to target the SPT effort not by focusing on the overall security problem confronted by the underlying system to which SAFIRE is added, but by looking within SAFIRE's distinguishing characteristics for ***security-relevant challenges that either arise because of, or are exacerbated by, the ways in which the SAFIRE augmented system is different to the non-SAFIRE system***.

What we discovered is that the SAFIRE augmented system, and in particular the considerable array of new third party components each with their own approach to security made the environment chaotic. Certainly the diversity of components and their approach to security was perhaps the defining challenge that arose because of or was exacerbated by SAFIRE.

(recall that SAFIRE is an enhancing *add-on* to an existing system)

To identify appropriate SPT objectives of the SAFIRE Project we sought to define an appropriate scope for the SPT effort. This effort is not intended to result in a comprehensive implementation of SPT as an outcome of the project. To pursue such a comprehensive solution would be ill-advised for several reasons.

1. It would be impossible to implement *di nuovo* all of the components of a comprehensive SPT solution because of the effort and cost, and it would be foolish to do so in any event, because there are abundant choices among both open source and proprietary solutions for most aspects of the needed security features.

2. It is unreasonable to expect that existing enterprises will abandon their investments in their own existing IT security solutions.

3. Security is an invasive discipline and it is not possible to "add on" security mechanisms to an existing system that already has security without reconfiguring the security measures that are already there to share or delegate control. The IT departments with existing security approaches will not easily abandon their stable solutions, and in particular will not be willing to (nor be expected to) modify their existing production infrastructure to depend upon a research prototype.

It would be a worthy goal to ultimately reduce the chaos of myriad access control solutions by providing one that could replace many of them. Thus we sought to provide a unifying policy and access control approach that would be sufficiently flexible and fine-grained to be a potential replacement for numerous diverse mechanisms currently employed in production systems, or in the combination of a production system with a SAFIRE add-on.

Consequently we identified a two-pronged approach:

- implementing the NGAC policy tool and policy server to make inroads on the problem of system policy specification and configuration, while delegating routine security issues to features already provided by a combination of the operating environment and chosen third party security products implemented by the IT department;

- mapping from the IISF to the Abstract Platform, intended to identify the implementation of those building blocks in the actual deployment (or to identify the assumptions that establish requirements on the environment in a deferred deployment).

2.4.2. Fulfillment of the Requirements

The use case requirements and how they are fulfilled by the functionalities represented in the Full Prototype and the supporting IT environment are summarized in Table 1. We allocate to the "abstract platform" (AP) the security features assembled by the IT department in the given base system, or to "NGAC" those addressed by the FP implementation. Many of the use case conventional security requirements are already addressed by the IT operating environment.

It is appropriate and expected that many security requirements originating from industrial use cases, which are based on existing implementations and infrastructure, will be fulfilled by the abstract platform that is a use case's operating environment. The Abstract Platform is "implemented" by the final mapping of the necessary IISF functional building blocks to the elements of a specific deployment on a specific OE with particular selection of 3PSM. This is accomplished according to guidelines provided by the methodology and subject to a risk analysis of the specific environment.

Abbreviations used in Table 1 are: OE – Operating Environment (OE = OS + HW), 3PSM – Third Party Security Mechanisms, AP – Abstract Platform (AP = OE + 3PSM), EP – Early Prototype, FP – Final Prototype, UC – Use Case, App – Application, SPTM – Security, Privacy and Trust Methodology/Mapping, Impl – Implemented, DSI – Deployment-Specific Implementation.

Table 1: Overview of use case requirements and how they are fulfilled

No.	Requirement	Overall Priority	Status
U94	Ensures data integrity	SHALL	Delegated to the AP
U95	Prevents unauthorized access	SHALL	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation DSI
U96	Provides secure access to generated knowledge in the cloud	SHALL	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation DSI
U98	Provides at least the same level of security as afforded by the operating environment	SHALL	Level commensurate with that afforded by OE, cannot provide greater
U99	Support for different user roles for secure access control	SHALL	Impl by NGAC with AP support
U100	Supports established authentication capabilities (hardware / passwords…)	SHALL	Delegated to the AP
U101	Provides data confidentiality management facilities	SHALL	Impl by NGAC FP with AP support, subject PEP/RAPs and illustrative UC adaptation
U102	Provides secure connectivity from machines to cloud	SHALL	Delegated to the AP
U103	Provides support for VPN connectivity	SHOULD	Delegated to the AP
U104	Provides encryption of data at rest locally	SHALL	Delegated to the AP
U105	Provides encryption of data at rest in the cloud	SHALL	Delegated to the AP
U106	Provides encryption of data during	SHALL	Delegated to the AP

	transport		
U107	Supports the establishment of rules/policies of trustworthiness (safety, privacy, security, reliability, resilience)	SHALL	Impl by NGAC FP and the SPTM
U108	Allows sharing of analysis results with designated people	SHOULD	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation DSI
U109	Allows collaboration of designated groups for building a knowledge base	SHOULD	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation DSI
U110	Authenticates who/what is sending data	SHOULD	Delegated to the AP

3. General Integration of NGAC

A single NGAC server instance can hold multiple policies simultaneously and can be switched among them or a single policy, or family of policies, can be designed to cover multiple domains. Such a policy can employ the NGAC server's capabilities for policy composition or its ability to operate under a special "all" policy that automatically composes all of the loaded policies in a particular way that is subject to some configuration options. The family of policies loaded and interpreted together in an "all" mode, must be written with the particular semantics of the "all" composition and the configuration settings in mind. This approach permits different protection domains to be created according to the placement of subjects and objects within the jurisdiction of one or more of the policies to be combined.

Another approach, when there are multiple independent protection domains each of which should be subject to its own policy, such as in a large distributed system, multiple instances of the NGAC server can be run to serve the clients that are subject to the distinct policies of each domain. The NGAC server provides a capability for declaring global policies that govern references between domains.

Furthermore, numerous potential applications of NGAC, in addition to the primary mission of endpoint data protection access control, may be possible within the IISF catalogue of security functional building blocks. As future SAFIRE installations are planned within existing manufacturing scenarios, it is suggested that the Industrial Internet Security Framework (IISF) be used to assess the completeness of coverage of the security needs that it identifies by the security measures provided both before the deployment of SAFIRE by the existing IT/OT[2] environment, and afterwards taking into account the addition of the SAFIRE security policy language and enforcement approach. In the process of performing such assessment an organization may consider the application of NGAC in additional ways within the infrastructure, that is consistent with the organization's needs, to achieve an increasingly common approach to policy enforcement.

Appendix D elaborates on ways in which NGAC can be used to support IISF-identified security functions. In this section we focus on the general integration of NGAC-aware client applications and in the following sections on integration with particular subsystems of the SAFIRE infrastructure.

3.1. NGAC-aware Client Applications

An NGAC-aware client application is one that has been adapted to access its controlled resources through an NGAC policy mediated interface. This is accomplished by modifying the application to replace direct resource access methods with calls to an NGAC policy enforcement point (PEP).

Our approach to the NGAC Functional Architecture has been to "unbundle" the PEPs and RAPs from the NGAC core infrastructure and to provide a policy server API that

[2] We employ here the terminology of Information Technology (IT) and Operational Technology (OT) that the IISF uses to describe the federation of enterprise information systems with industrial process control systems in modern factories.

can be called by custom PEPs that are appropriate to the kinds of resources and how they are used. This approach places all of the steps needed for the adoption of NGAC within the application developer's domain and control.

The Client Application and the PEPs and RAPs for the resources it accesses are all developed by the application developer according to a simple architectural pattern referred to as the CA/PEP/RAP pattern.

3.2. CA/PEP/RAP Architectural Pattern

Figure 3 focuses on the Resource Access Path first identified in the NGAC functional architecture of Figure 1. Client applications (CA) of the NGAC-managed object space must request access to objects (resources) through a Policy Enforcement Interface provided by a Policy Enforcement Point (PEP). The developer of the CA must create a PEP and a RAP for the objects required by the CA, if such a PEP and RAP do not already exist for objects of that kind, in order to complete the resource access path. The architectural pattern to be used is very straightforward but should be followed faithfully if the integrity of the access control is to be maintained.

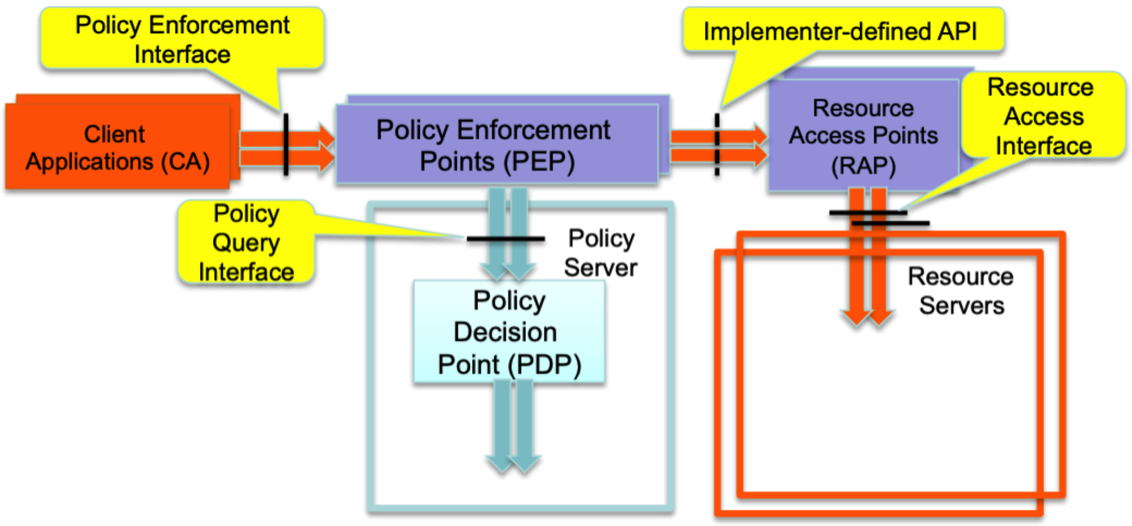

Policy Enforcement Interface
Implementer-defined API
Client Applications (CA)
Policy Enforcement Points (PEP)
Resource Access Points (RAP)
Resource Access Interface
Policy Query Interface
Policy Server
Policy Decision Point (PDP)
Resource Servers

Figure 3: CA/PEP/RAP pattern forming the resource access path

A call is made by the PEP to the Policy Decision Point (PDP) in the policy server to determine whether a particular access (consisting of: user, access right, object) is permitted under the policy before the PEP calls the appropriate RAP to access the resource. This call to the PDP may even be stubbed out (returning 'grant') during development or in the absence of an operating policy server so that the developer's work is not hindered. Alternatively, the policy server may be places in a mode in which it will respond 'grant' to all access queries without a policy in place. This case corresponds to the accessing of the resource by the client without access control. A corresponding mode in which all access queries return 'deny' completes the ability to test the resource access path for grant and deny conditions. The developer is encouraged to develop RAPs that use the same resources and resource servers, hence the same

Resource Access Interface that the application would have used in an unmediated scenario.

The CA/PEP/RAP architectural pattern is designed to achieve security through two essential characteristics. First, the PEP operates in a distinct process from that of the CA, able to be executed under an identity distinct from that of the user on behalf of which the CA is operating. The same is true for the RAP. Second, the PEP and the RAP are very simple in their purpose and are to be kept very simple in their implementation. Thus they are relatively easy to assure by inspection of their code and of the permissions given to their executable programs.

The PEPs and the RAPs, as well as the NGAC server itself, execute as a distinct 'ngac-user' identity within the host operating system (OS). The intrinsic access control mechanisms of the underlying platform OS are used to segregate all NGAC-managed objects and to provide access to these objects exclusively to processes executing on behalf of the 'ngac-user' identity.

Finally, in a deployed industrial scenario, where there are enterprise-critical resources and potential accessibility by untrusted users and agents, and communications among the components of the NGAC functional architecture, the communications among the CA, PEP, PDP, and RAP should be configured to be performed over encrypted and authenticated sockets. This step requires an additional layer of administration, including certificate management, key generation, distribution, and management. These are burdensome but well understood activities in practice and for this reason were not done in the prototype.

3.3. PEP/RAP DESIGN, IMPLEMENTATION AND OPERATION

3.3.1. PEP/RAP Unbundling Rationale

The PEP and the RAP are "unbundled" from the core implementation of the NGAC functional architecture as shown in Figure 1. In this way the efforts of the developers of Client Applications (CA) can be decoupled from those of the implementers of the NGAC core. In a reference implementation, the PEP and RAP were an integral part of the NGAC implementation. This had a some adverse consequences: the (complex) implementation needed to be modified and redeployed for each new kind of resource, and the modifications of client applications needed to conform to an NGAC-mandated, and constraining, policy enforcement interface (PEI) available only in one programming language. By unbundling the PEP and RAP the policy enforcement interface is left to the discretion of the developer and the protected resource can be accessed at an appropriate level of abstraction chosen by the developer.

It is instructive to consider the differences between an unmediated access scenario and the NGAC-mediated access scenario. Suppose an application had been developed to use resources directly with no access control mediation. The application accesses the resource through an interface provided by the resource server[3]. When NGAC mediation is introduced, the code fragment (or a similar code fragment) that previously directly

[3] The resource server implements (or consolidates) and presents the operations that may be performed on the resource.

accessed the resource is reproduced within the logical perimeter of the access control system and uses the existing resource access interface (RAI). The code in the Application (now an NGAC Client Application) is replaced with code that uses the NGAC-provided Policy Enforcement Interface to access the now Protected Resource. If the policy enforcement interface provided by NGAC is not well suited to the Application and the Resource this conversion activity can be disruptive to the application developer, possibly requiring restructuring or refactoring of the application. This will occur, for example, when object-oriented constructs are used to encapsulate the implementation of objects.

To avoid this difficulty, the present implementation of NGAC takes the approach reflected in the functional architecture of Figure 1, whereby the PEP and RAP are factored out of the rest of the NGAC implementation (here the Policy Server).

3.3.2. Adapting the Client Application

Referring to Figure 3, generally, when one takes code that directly references a resource access interface out of the application and replaces it with NGAC PEP interface references, the removed code will be represented in some form in the RAP portion of NGAC as shown in Figure 4.

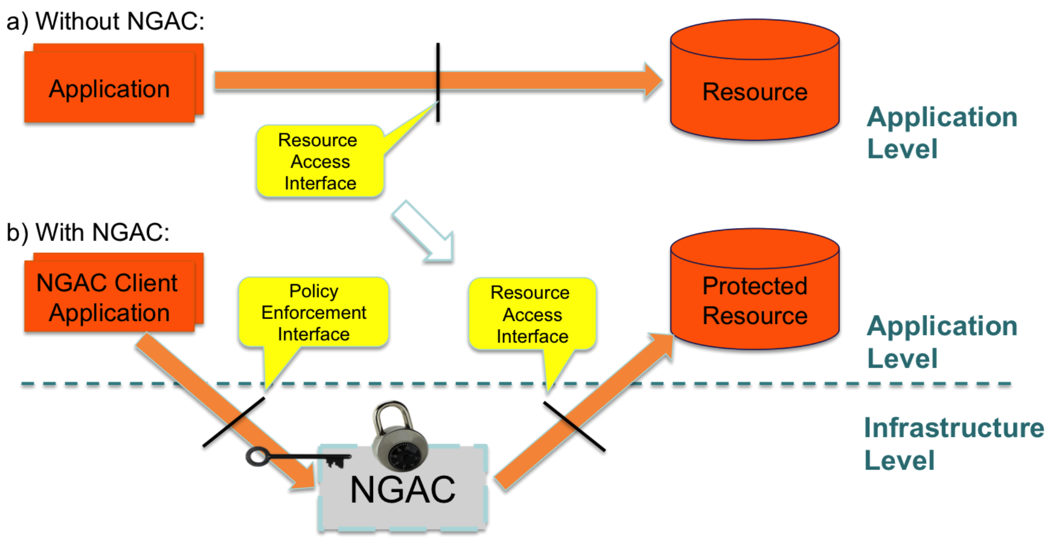


Figure 4: Adaptation of an application to use NGAC

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 one-to-one. 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.

3.3.3. Design of the PEP/RAP

The Client Application developer must create the PEP and the RAP to complete the resource access path (shown in orange in the figure). Now instead of the developer having to adapt to the constraints of a fixed Policy Enforcement Interface for all resource kinds, the developer is free to define the Policy Enforcement Interface when implementing the PEP. This can be tailored to be closer to the original access pattern. Further, it puts the entire mediated resource access path in the control of the developer.

In security terms, the PEP and RAP are part of the NGAC Trusted Computing Base. Though an application developer may not ordinarily implement part of the trusted computing base, they are expected to implement the PEP and RAP if an appropriate PEP and RAP do not already exist. There are guidelines for the construction of the PEP and RAP to make the developer's task easier. The PEP and the RAP should both be small and logically simple executable programs, and may possibly exist within the same executable program. However, it is important that they can be granted privileges distinct to those of the client application. The RAP, which accesses the resource access interface, consists essentially of the code that previously appeared in the Application to access the resource, that was replaced by the call to the policy enforcement interface. The control flow of the PEP is basically a conditional statement that invokes the PDP access query function as the condition of the branch. The access call specifies the user, the object, and the operation to be performed on the object. Using the server's current policy, the PDP computes whether the user is permitted to perform the operation on the object and returns either "grant" or "deny". The "grant" branch of the PEP branching statement invokes the RAP to perform the specified operation on the object, while the "deny" branch immediately returns a failure to the CA.

The simplicity of the PEP and the RAP, and their isolation as separate and distinct execution units, are the essential foundation of the strategy of unbundling them from the core NGAC implementation. Since they are simple, they can be verified by inspection, and thus trustworthiness may be established. Since they are separate execution units, they have well-defined interfaces and cannot be corrupted by the CA or by other processes. Finally, as distinct execution units (processes) they can be given privileges distinct from those of the CA. Specifically, they can be configured to have exclusive access to the protected resources (or resource servers), so that there is no way for the CA or another process to bypass them and to directly access the protected resource.

Table 2 uses informal pseudocode to illustrate the conceptual simplicity of the PEP and the RAP. Of course, since both may be implemented as RESTful services, there will be the necessary detail of marshalling arguments and invoking the PEP and the RAP and returning and interpreting results, but following this simple design pattern will help to assure by inspection that these are trustworthy components of the NGAC resource access path.

Table 2: PEP and RAP pseudo-code

PEP pseudo-code	RAP pseudo-code

`pep( Op, Object, Data )`
 	
`determine User/Session from the environment`
`query_result = pdp:access(User, Op, Object)`
`if query_result == 'grant' then`
`  ObjInfo = pdp:getobjectinfo(Object)`
`  ra_result = rap(Op, Object, Data, ObjInfo)`
`  return to Client App: ra_result`
`else`
`  return to Client App: 'Op on Object denied'`
	`rap( Operation, Object, Data, ObjInfo )`
	
	`identify res_server using ObjInfo`
	`result = res_server:Operation(Object, Data)`
	`return to PEP: result`

In the PEP pseudo-code we first determine the identity of the User/Session associated with the call to the PEP by the CA. This may be done in a variety of ways depending on details of the local environment, available identity services, keyed mutually-authenticated encrypted communication channels, etc. Then the ***access*** API of the PDP is invoked to obtain a decision whether the User/Session is permitted to perform the Op on the Object according to the currently active policy. If the PDP responds with 'grant' then the PEP proceeds to get the metadata of the object, if necessary, including for example the location or resource server that holds the data, and then the PEP calls the RAP to perform the Op on the Object, optionally using the Data. The result is reported to the CA. If, on the other hand, the PDP query result was 'deny' (or anything other than 'grant') then the PEP does not perform the operation and it reports the access denial to the CA. The essence of what the PEP is trusted to do consists of these things: to consult the PDP, to *not perform* the requested operation if the PDP responds 'deny', and to request the correct operation on the correct resource if the PDP response is 'grant'.

The RAP pseudo-code simply identifies the appropriate resource server using the ObjInfo argument. This step may be unnecessary if the PEP and RAP are specific to a resource server. The RAP then invokes the resource server to perform the Operation on the Object, optionally using the Data. The RAP returns the result of the Operation to the PEP.

The PEP and RAP operate with identities that afford them exclusive access to the associated resource servers, so that they cannot be bypassed. In some cases it may be convenient to package the PEP and RAP together in the same process, or even to package the PEP, RAP and resource server together in the case that the protected resource itself is a web service or a database management system. Nonetheless, the security properties of resource access path are improved if the trusted PEP and RAP components are not packaged with a more complex, and therefore less trustworthy, web service or DBMS, but rather serve as a proxy for the more complex service.

The unbundled PEP/RAP arrangement is convenient for the CA developer because the development of the CA, PEP, and RAP can be done independently of development and maintenance of the Policy Server. In fact, the server provides a feature that facilitates testing of the CA/PEP/RAP path. By executing the server with a special command line option, the server can be made to always return "grant" or "deny" to all access queries regardless of the loaded policy or even if there is no loaded policy. By testing with a local instance of the server with either of these options, even before the policy has been written in the policy language, it is possible to test both the access granted and access denied paths of the CA/PEP/RAP. Starting the server in “grant mode” the resource access path should operate just like the original resource access pattern in which the Application had unrestricted use of the resource. “Deny mode” can be used to test the CA/PEP’s error handling execution paths. It is also possible to set the server to grant mode or deny mode by calling the ***setpol*** API with the argument ‘grant’ or ‘deny’.

3.3.4. PEP Policy Enforcement Interface (peapi)

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

peapi/getobject – return object content (read)

Parameters

- object = <object identifier>

Returns

- “success” or “failure”
- <object data>

peapi/putobject – set object content (write)

Parameters

- object = <object identifier>
- content = <object data>

Returns

- “success” or “failure”

3.3.5. RAP Resource Access Interface (raapi)

Resource Access Points embody access methods that would likely be used directly in the client application if it were to access the resource without mediation by NGAC. The RAPs may be specific to a kind of object and may be very similar to a code fragment that would have been used in the application under such a scenario. Thus, the details of the Resource Access Interface may be chosen by the application developer, and encapsulated in a small distinct process. The PEP determines the appropriate RAP to call after asking the PDP for a policy verdict and getting the object’s metadata from the PIP through a PQI call (***getobjectinfo***).

3.3.6. PEP and RAP Implementation Templates

Templates for implementation of the PEP and RAP that provide examples of invoking the Policy Decision Point (NGAC server) and Resource Access Points by Policy Enforcement Points are included in the distribution.

The PEP template exhibits the definition of a server for a RESTful Policy Enforcement Interface providing 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 is a example of a RAP for ordinary OS files, and currently implements file_open, file_close and file_read.

The implementations of the PEP and RAP templates are contained in the following Prolog files:

- pep.pl – defines an example PEP server for a policy enforcement interface
- rap.pl – defines a simple example of a RAP for ordinary files

3.4. 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 a non-benign environment, if they can operate without malicious interference. That is, in a production environment that is exposed to real threats the functional architecture must be affirmatively *enforced*. NGAC is a reference validation mechanism (RVM) and 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 essential properties of a reference validation mechanism[4] (besides the obvious first property: correctness), that is, it must be tamper-proof and non-bypassable (“always invoked”). These properties cannot be achieved by the RVM itself, but must be provided for the RVM by its environment through a proper architectural 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 domain. 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 for the demonstration. 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 level of robustness in a deployment. We have found this to be the case of research projects in

[4] As per the “Anderson Report” of 1972.

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 domain 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 ssume a Unix-like operating system or one providing similar features to the basic protections mentioned above.

First we outline the requirements: there must be isolation of the protected resources and a trusted chain of execution for components with the ability to access those isolated resources.

Specifically, and at a minimum, the PEP, the RAP, or a combined PEP/RAP, and the PDP/PAP/PIP should run as distinct processes that should be run with a distinct 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 or PEP/RAP through the corresponding resource server. For example, for file objects in a Unix-like system all of the files placed under the jurisdiction of NGAC should be made to be read and write accessible *only* to the user *ngac*. Further, the executable file for the RAP or PEP/RAP should be configured to have the user id *ngac*, and have the set-user-id-on-execution attribute set. This will result in read and write access to the objects being given to the PEP, which is trusted to responsibly use this access only according to the responses of the PDP. Finally, the PEP and RAP executables should be configured to be executable only by the system shell or startup script to prevent an unauthorized subject from starting them and potentially gaining access to the protected resources while acting as a *confused deputy*.

3.5. Deploying the NGAC components

The ngac server does not need to be running in any one particular place, but it should be accessible to be used by all PEP/RAPs. Though it is not a strict rule, generally, running the PEP close to the client application and running the RAP close to the resource makes sense, unless the PEP and RAP are combined, in which case a decision must be made whether to deploy it closer to the client application or to the resource. If multiple CAs access the PEP from different locations and it does not make sense to deploy multiple instances of the PEP, then the combined PEP/RAP can be run close to the resource that it controls.

3.6. Initiating the NGAC components

To achieve the enforcement of the NGAC functional architecture discussed above, it is important to consider how the NGAC components are initiated, however they may be deployed. The particulars of the solution are very dependent on the organization of the NGAC-using subsystems and application, and on the characteristics and conventions of the IT environment in which NGAC is deployed. We outline the general principles for consideration by NGAC adopters.

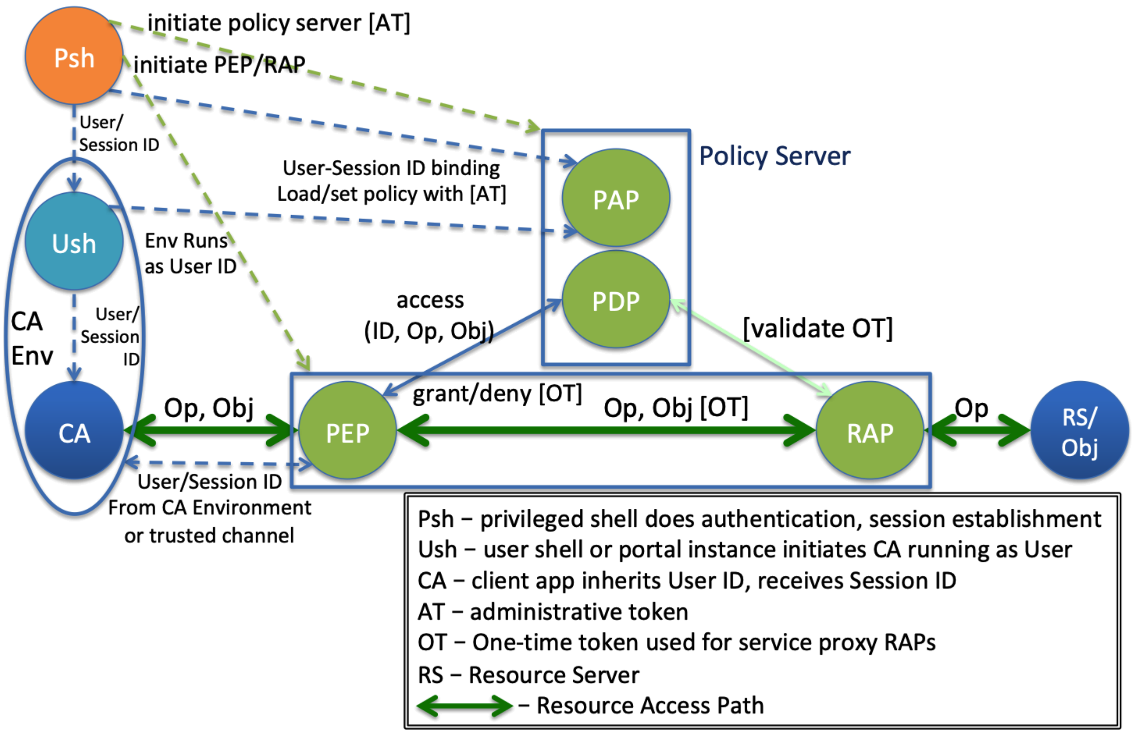

Psh
initiate policy server [AT]
initiate PEP/RAP
User/ Session ID
User-Session ID binding
Load/set policy with [AT]
Policy Server
PAP
PDP
Ush
Env Runs as User ID
CA Env
User/ Session ID
access (ID, Op, Obj)
[validate OT]
CA
Op, Obj
PEP
grant/deny [OT]
Op, Obj [OT]
RAP
Op
RS/ Obj
User/Session ID From CA Environment or trusted channel
Psh – privileged shell does authentication, session establishment
Ush – user shell or portal instance initiates CA running as User
CA – client app inherits User ID, receives Session ID
AT – administrative token
OT – One-time token used for service proxy RAPs
RS – Resource Server
– Resource Access Path

Figure 5: Initiation of the NGAC functional architecture components

The general NGAC initiation scheme is depicted in Figure 5. The legend provides the meaning of abbreviations used in the figure. NGAC components are depicted with their customary acronyms.

The privileged shell "Psh" is a trusted program or script that is privileged to spawn user shells "Ush" and to set the user id under which they operate. This shell is running, or is started by, the system initialization script. This shell starts the NGAC infrastructure, including initiating the policy server, the PEP and the RAP, as well as the environments that run the NGAC client applications.

The Psh also has the responsibility to perform or delegate certain NGAC administration functions (by way of passing the Administrative Token), including the establishment of an NGAC session ID corresponding to the user ID for a particular instance of a Ush, and the administration (load, combine, set, etc.) of the policies to be used for the clients to be spawned by the Ush. Depending upon the organization and constraints provided by the Ush, the Psh may delegate all or part of these administrative responsibilities to the Ush by passing the AT to it.

The user shell “Ush” is a program or a script that initiates the client application. These is an instance of the Ush per user session. This may be a “portal” kind of web application that runs different client applications in response to user interactions. It could also be an “orchestrator” of client applications. As long as the Ush limits the ability of the user to invoke only a controlled collection of functions (client applications) and cannot be caused by the user to abuse its possession of the AT, then it is a candidate to have session and/or policy administration delegated to it.

The client application “CA” accesses an object “Obj” provided by a resource server “RS” by making a request to the PDP to perform an operation “Op” on Obj. To query a policy a user or session ID is needed. The PEP determines the user ID or the session ID according the manner in which it is invoked or the channel by which the CA query is made. If through an encrypted channel, then the mutual authentication performed to establish the channel will provide the ID. PEPs can also be poly-instantiated, that is, an instance of a PEP per Ush, per application (allowing the application is to be considered the “user” from the standpoint of policy), per session, or per user. The PEP subsequently uses the user/session ID in requests made to the policy decision point “PDP”.

The PDP consults the current policy to return a ‘grant’ or ‘deny’ response, and, in the event that the object type indicates that the RAP is a web-service proxy using the NGAC protocol for web services, the PDP will also generate and return a one-time token “OT”. This token will be passed to the RAP by the PEP, which will in turn validate the OT with the PDP before passing control to the web service with the requested operation. In any event, the RAP performs the requested operation and returns the result to the PEP to be relayed to the CA.

Figure 6 illustrates the sequence of actions among the NGAC components for an operation OP on an ordinary object o2 performed by a client application running as user u1. The Master Init in this chart represents the Psh in the previous figure.

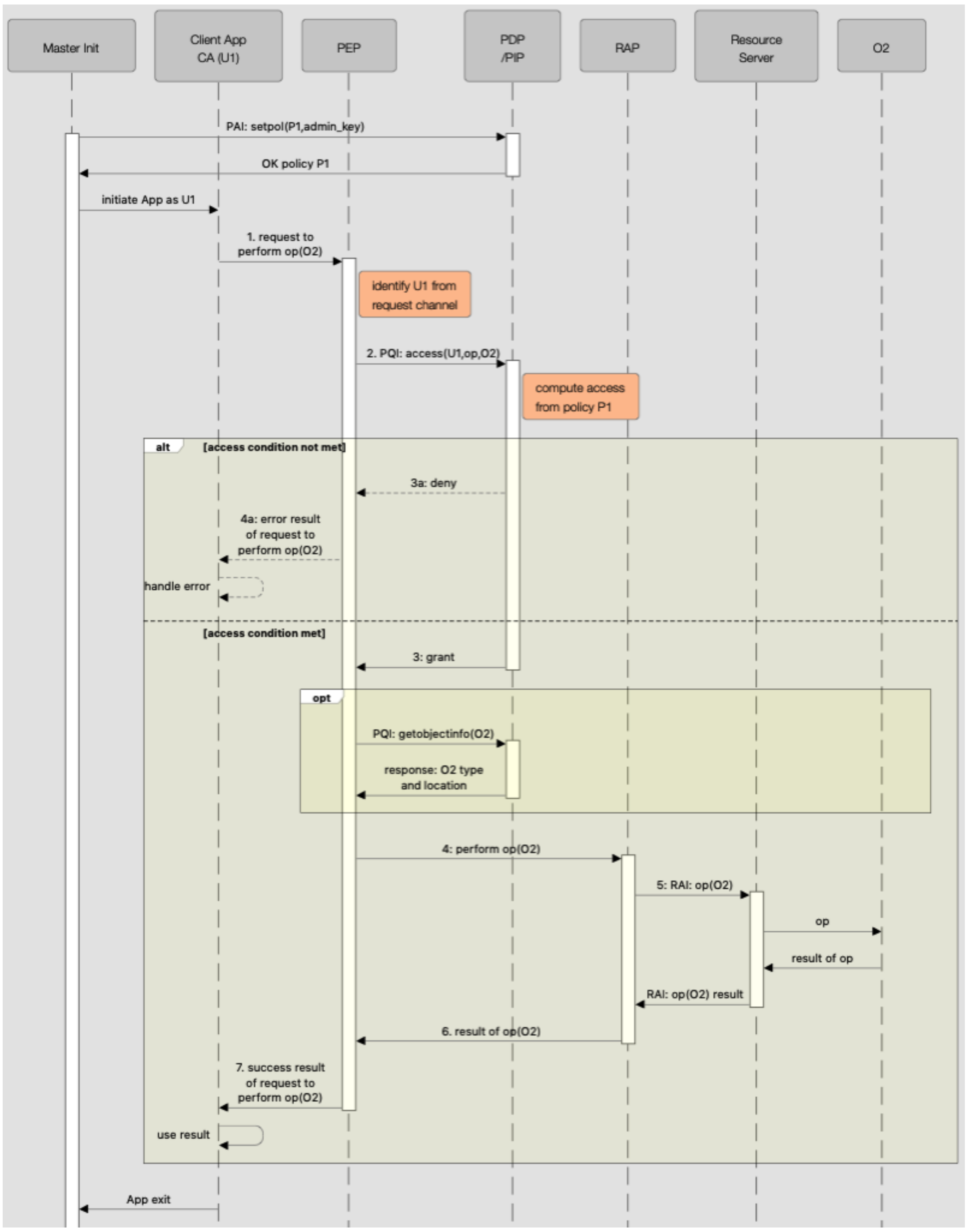


Figure 6: Sequence of actions for NGAC mediation of ordinary object operation

3.7. WEB SERVICES AS PROTECTED RESOURCES

As mentioned in the preceding section, web services may be the objects of an NGAC policy. There are two basic ways to approach this, but there are many possible variations. In the first case the Policy Enforcement Point for a web service is separate from the service and it may not know that the requested operation is a web service call until it is informed by the PDP and it passes the request to the proxy. In the second case the PEP/RAP is the public proxy and it calls the PDP itself.

We illustrate the first case, which is supported by the OT scheme of the previous section. The Policy Enforcement Point discovers that the requested object operation is a web service after it makes the access query and then requests from the PDP the detailed information on the referenced object. In this case, the PDP can be configured to return a one-time token (OT) associated with the successful ('grant') PDP response. This token is passed to the RAP that acts as a proxy for the Web service and the RAP will validate the token for the requested operation with the PDP, after which the PDP forgets the OT, and the proxy calls the real Web service to carry out the operation. Figure 7 shows the sequence of interactions for such a transaction; the client Invokes service A, the PEP creates the OT (represented here as nonce N) and returns it to the PEP, N is passed along with the request to the Proxy for A, which validates N with the PDP, if this succeeds then the Proxy invokes service A which returns the result to the PEP for relay to the Client.

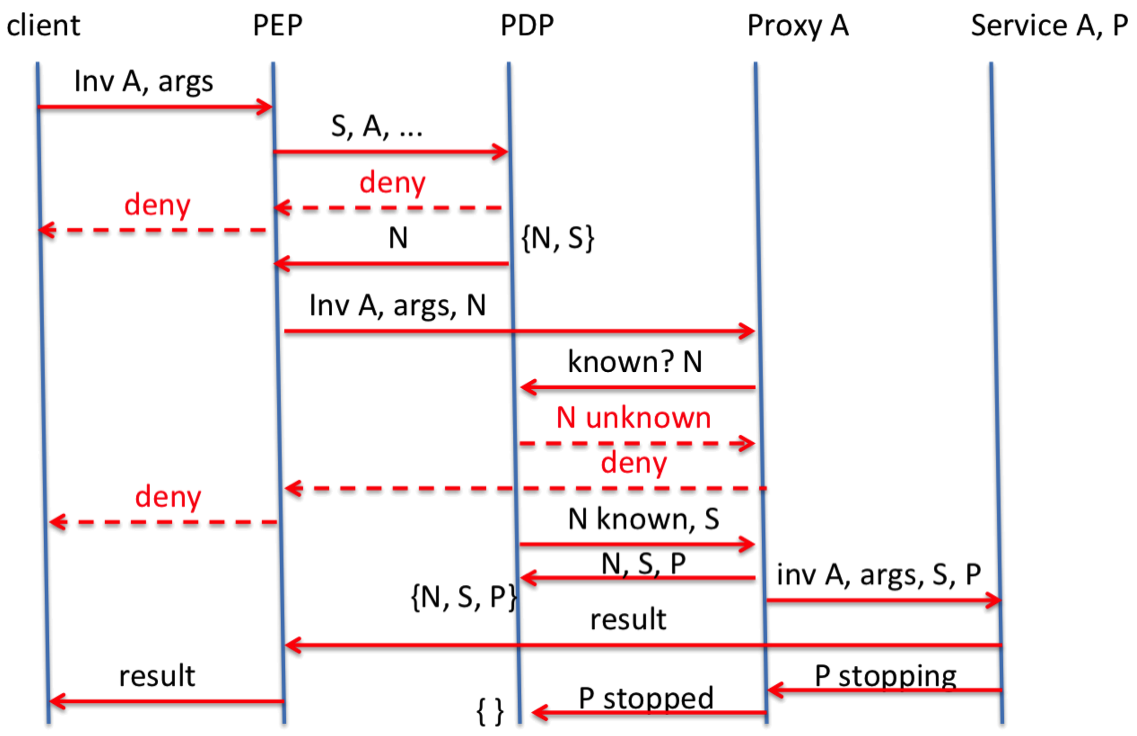


Figure 7: Sequence for NGAC-mediated Web service proxy

4. Integration with Predictive Analytics

The Predictive Analytics Platform allows SAFIRE users to do advanced analytics in real time, storing huge amounts of data and using web visualization tools to easily query and visualize the stored data, as described in deliverable D2.4 Full Prototype of Predictive Analytics Platform. Moreover, the Predictive Analytics Platform offers different web services for interacting with different modules. An architectural overview of the described platform can be seen in Figure 8.

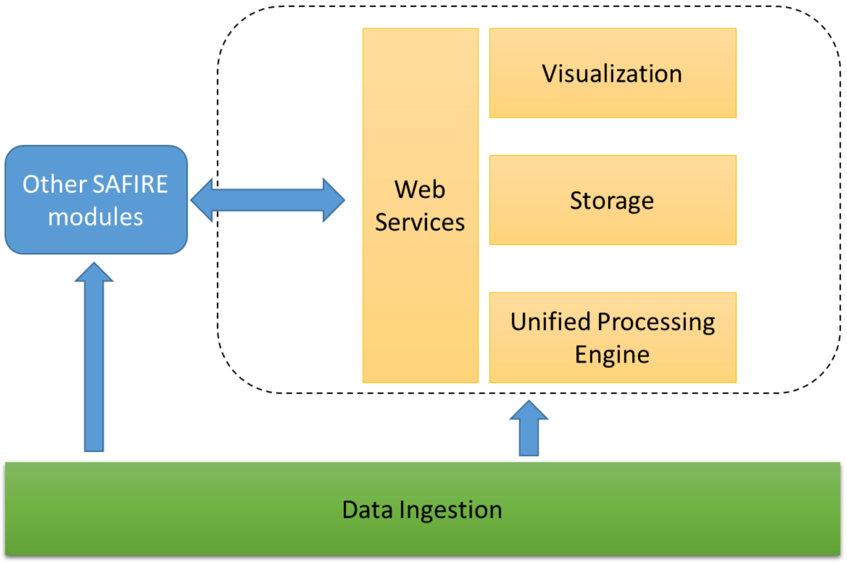


Figure 8: Conceptual Predictive Analytics Platform architecture

For each of the services offered by the Predictive Analytics Platform there are different components that may need to be secured. In the following we describe the standard industry security features associated with the services and components. For these services and security features we consider their attributes and how they apply to the SAFIRE security requirements. We indicate the areas where advantages are provided through integration with the Security, Privacy and Trust Framework developed in SAFIRE.

4.1. Storage

The relational storage is covered using PostgreSQL, a leading Open Source Relational Database System. The relational database performs several of the data quality checks listed in D2.5 Final Specification of Predictive Analytics Platform. The No-SQL storage is provided by using the Apache Cassandra Database, a key-value based database with horizontal scalability properties and with great integration with the Big Data landscape via different connectors. The following summarises the integration points for the SAFIRE SPT Framework and the Storage Layer of the Predictive Analytics Framework.

4.1.1. Cassandra Distributed Database

Cassandra database is used to store product factory data in a scalable way within SAFIRE, mainly storing time series data form different sensors. This Database is used in production in companies such as APPLE, Netflix and PayPal among others providing a lot of security features that are overviewed in the next sections.

There are three main components to the security features provided by Cassandra:

- TLS/SSL encryption for client and inter-node communication
- Client authentication
- Authorization

By default, these features are disabled as Cassandra is configured to easily find and be found by other members of a cluster. In other words, an out-of-the-box Cassandra installation presents a large attack surface for a bad actor and therefore in a production environment those features should be configured accordingly.

TLS/SSL Encryption

Cassandra provides secure communication between a client machine and a database cluster and between nodes within a cluster. Enabling encryption ensures that data in flight is not compromised and is transferred securely. The options for client-to-node and node-to-node encryption are managed separately and may be configured independently.

SSL Certificate Hot Reloading

Beginning with Cassandra 4, Cassandra supports hot reloading of SSL Certificates. If SSL/TLS support is enabled in Cassandra, the node periodically polls the Trust and Key Stores specified in `cassandra.yaml`.

Inter-node Encryption

The settings for managing inter-node encryption are found in `cassandra.yaml` in the `server_encryption_options` section. To enable inter-node encryption, change the `internode_encryption` setting from its default value of `none` to one value from: `rack`, `dc` or `all`.

Client to Node Encryption

The settings for managing client to node encryption are found in `cassandra.yaml` in the `client_encryption_options` section.

Roles

Cassandra uses database roles, which may represent either a single user or a group of users, in both authentication and permissions management. Role management is an extension point in Cassandra and may be configured using the `role_manager` setting in `cassandra.yaml`. The default setting uses `CassandraRoleManager`, an implementation which stores role information in the tables of the `system_auth` keyspace.

Authentication

Authentication is pluggable in Cassandra and is configured using the `authenticator` setting in `cassandra.yaml`. Cassandra ships with two options included in the default distribution.

By default, Cassandra is configured with `AllowAllAuthenticator` which performs no authentication checks and therefore requires no credentials. It is used to disable authentication completely. Note that authentication is a necessary condition of Cassandra's permissions subsystem, so if authentication is disabled, effectively so are permissions.

The default distribution also includes `PasswordAuthenticator`, which stores encrypted credentials in a system table. This can be used to enable simple username/password authentication.

Enabling Password Authentication

Before enabling client authentication on the cluster, client applications are pre-configured with their intended credentials. When a connection is initiated, the server will only ask for credentials once authentication is enabled, so setting up the client-side config in advance is safe. In contrast, as soon as a server has authentication enabled, any connection attempt without proper credentials will be rejected which may cause availability problems for client applications. Once clients are setup and ready for authentication to be enabled, the following procedure is followed to enable it on the cluster.

Pick a single node in the cluster on which to perform the initial configuration. Ideally, no clients should connect to this node during the setup process, so you may want to remove it from client config, block it at the network level or possibly add a new temporary node to the cluster for this purpose. On that node, perform the steps elaborated in the Cassandra security document.[5]

1. Open a cqlsh session and change the replication factor of the `system_auth` keyspace. By default, this keyspace uses `SimpleReplicationStrategy` and a `replication_factor` of 1. It is recommended to configure a replication factor of 3 to 5 per-DC.
2. Edit `cassandra.yaml` to change the `authenticator` option to `PasswordAuthen-ticator.`
3. Restart the node.
4. Open a new cqlsh session using the credentials of the default superuser.
5. During login, the credentials for the default superuser are read with a consisten-cy level of `QUORUM`, whereas those for all other users (including superusers) are read at `LOCAL_ONE`. In the interests of performance and availability, as well as se-

[5] https://cassandra.apache.org/doc/latest/operating/security.html

curity, operators should create another superuser and disable the default one. This step is optional, but highly recommended. While logged in as the default superuser, create another superuser role which can be used to bootstrap further configuration.

6. Start a new cqlsh session, this time logging in as the new_superuser and disable the default superuser.
7. Finally, set up the roles and credentials for your application users with CREATE ROLE statements.

At the end of these steps, the one node is configured to use password authentication. To roll that out across the cluster, repeat steps 2 and 3 on each node in the cluster. Once all nodes have been restarted, authentication will be fully enabled throughout the cluster.

Authorization

Authorization is pluggable in Cassandra and is configured using the `authorizer` setting in `cassandra.yaml`. Cassandra ships with two options included in the default distribution.

By default, Cassandra is configured with `AllowAllAuthorizer` which performs no checking and so effectively grants all permissions to all roles. This must be used if `AllowAllAuthenticator` is the configured authenticator.

The default distribution also includes `CassandraAuthorizer`, which does implement full permissions management functionality and stores its data in Cassandra system tables.

4.1.2. PostgreSQL Relational Database

PostgreSQL database is used to store relational database along with data quality checks within SAFIRE.

As in the case of Cassandra, there are three main components to the security features provided by PostgreSQL:

- TLS/SSL encryption for client and inter-node communication
- Client authentication
- Authorization

By default, these features are disabled in PostgreSQL and therefore in a production environment should be configured accordingly.

TLS/SSL Encryption

PostgreSQL provides secure communication between a client machine and a database cluster and between nodes within a cluster. Enabling encryption ensures that data in flight is not compromised and is transferred securely.

Roles

PostgreSQL provides database roles, which may represent either a single user or a group of users, in both authentication and permissions management. PostgreSQL establishes the capacity for roles to assign privileges to database objects they own, enabling access and actions to those objects. Roles can grant membership to another role. Attributes provide customization options, for permitted client authentication.

Authentication

Authentication is pluggable in PostgreSQL. PostgreSQL by default provide different authentication mechanisms[6] by default.

4.1.3. NGAC Integration

The NoSQL Casandra database supports custom authentication and authorization mechanisms and therefore small adapters are applied for integrating the NGAC server. In the case of the relational PostgreSQL database the integration point for NGAC is with the supported authentication mechanisms. An alternative approach that will be explored in preparation for commercial exploitation is to incorporate the NGAC extension into PostgreSQL as it is available in open source.

4.2. WEB SERVICES

4.2.1. Feature set

The web services developed within SAFIRE Predictive Analytics module are mainly used to interact with the analytic models (e.g. getting predictions). The modules have been developed using an industry standard Framework called Spring Boot[7].

SpringBoot provides different modules for authentication and/or authorization purposes. The main Spring Boot component is Spring Security[8], a library for getting authorization and authentication for web services using multiple providers and methods such as stateless and stateful mechanisms.

4.2.2. NGAC Integration

The integration point with NGAC is through the REST API capabilities that enable the Spring Security module to support the NGAC server for both authentication and authorization.

4.3. VISUALIZATION

In order to be able to visualize the results of the analytics two tools are provided within SAFIRE Predictive Analytics Platform for different kind of users:

[6] https://www.postgresql.org/docs/current/auth-methods.html
[7] https://spring.io/projects/spring-boot
[8] https://spring.io/projects/spring-security#overview

1. **Business Intelligence**: in order to provide support for Business intelligence dashboards Apache Superset tool is provided. It provides an easy way to define different web-based dashboards with a lot of connectors to multiple databases.

2. **Data Scientist**: in order to provide support for data scientists that need to interact easily with the stored data in SAFIRE, to execute interactive advanced analysis over huge quantities of data, and to visualize those analyses in an easy way, Apache Zeppelin is provided within SAFIRE to fulfil this task. Zeppelin is a web-based notebook that easily provides support for interactive analytics over Big Data.

4.3.1. Apache Superset

Apache superset is a modern, enterprise-ready business intelligence web application that provides an intuitive interface to explore and visualize datasets. It has a wide array of beautiful visualizations that can be used to showcase data.

Security in Superset is handled by Flask AppBuilder (FAB). FAB is a “Simple and rapid application development framework, built on top of Flask.”. FAB provides authentication, user management, permissions and roles. The security documentation can be seen at https://flask-appbuilder.readthedocs.io/en/latest/security.html.

By default, Superset ships with a set of roles that are handled by Superset itself. You can assume that these roles will stay up-to-date as Superset evolves. Even though it’s possible for Admin users to do so, it is not recommended that you alter these roles in any way by removing or adding permissions to them as these roles will be re-synchronized to their original values as you run your next superset init command.

Since it’s not recommended to alter the roles described here, it’s right to assume that your security strategy should be to compose user access based on these base roles and roles that you create.

FAB provides authentication backends for standards like OAuth[9] or OpenID[10], however it can be easily extended for supporting custom approaches with of a little bit of code and configuration[11].

4.3.2. Apache Zeppelin

Apache Zeppelin has different mechanisms to protect both the developed notebooks and to secure the communication between the servers and the clients. Apache Zeppelin uses Apache Shiro library[12] to provide authentication and authorization mechanisms. Apache Shiro is a modular security library that can be easily extended and that have a lot of industrial security features included.

[9] https://oauth.net/2/
[10] https://openid.net/
[11] https://medium.com/@mmutiso/authenticate-apache-superset-with-a-custom-user-store-c4511ab0f798
[12] https://shiro.apache.org/

An overview of the authorization and authentication features provided out of the box can be seen on the following webpages.

- https://shiro.apache.org/authentication-features.html
- https://shiro.apache.org/authorization-features.html

4.3.3. NGAC Integration

Both Superset and Zeppelin visualization tools for the SAFIRE Predictive Analytics module support custom authentication and authorization mechanisms so the integration points with NGAC are small adapters that make it possible to use the NGAC server with the authentication and/or authorization features.

4.4. Unified Processing Engine

The Unified Processing Engine provides support for doing advanced analytics on both real-time and batch approaches. This module is based on Apache Spark. A Unified Big Data Framework. Moreover, as defining complex real-time analytics is difficult right now with this kind of framework a Complex Event Processing (CEP) engine has been included on the platform to cover this use case. The CEP engine used on SAFIRE is called Espertech. Espertech, provides to the developer a Domain Specific Language (DSL) language based on SQL that helps to define complex real time analytics patterns.

Both engines are widely used and support industry standard network communication via SSL. The unified processing engine does not need to integrate with the NGAC server. However, the programs that run in the engine are required to pass the credentials to the storage engine queried in order to load or store the desired data.

5. Integration with Reconfiguration and Optimisation

Section 6.4.5 of D3.5 provides a description of the implementation of the SAFIRE security framework (SSF) in the Reconfiguration and Optimisation Engine (OE). The following discussion is reproduced from there.

The Optimisation Engine (OE) container can be invoked only by the SAFIRE Situation Determination (SD) module, authorised earlier following the SSF. Similarly, the connection between OE and Objective Function (OF) uses TLS and, when it exists, the connection between OF and Predictive Analytics (PA) does as well. Notice that OE and OF containers are generated independently for each end user, thus there is no possibility of accessing other user data as long as end users do not reuse the same certificates or keys.

OE is planned to use TLS certificates on both the server and the client side to provide a proof of identity. Consequently, it requires both the client and server to own a certificate signed by a specific Certificate Authority (CA), for example using OpenSSL. This way of securing the connection is widely available, supported by Docker and the majority of public cloud vendors. Similarly, Kubernetes supports TLS and even each Kubernetes cluster has its own root CA that can be used by the cluster components to validate the clients and servers' certificates. If deployed to a Kubernetes cluster, OE, OF and PA can request a certificate signing using the certificates.k8s.io API. Similarly, a root CA (named AWS Certificate Manager - ACM) is available when using Amazon Web Services (AWS). TLS is also supported by the Network Load Balancer, which can be used with both EC2 and Fargate Launch Style. TLS is also available in IBM Cloud, Azure, OpenStack and other public clouds so choosing this protocol does not limit the future deployment options. The independence of the execution containers and of the data they employ (subject to disciplined use of access keys, explained later in this section), along with the guarantee that the above-enumerated containers can be invoked only by a trusted invocation path, covers the architectural integrity aspect of security with the combination of communication integrity.

The Next Generation Access Control (NGAC) methodology and configurable policy, as described in SAFIRE deliverable D5.5 [SAFIRE D5.5, 2018], are responsible for the above-mentioned *discipline in the use of access keys*. The components of the Reconfigurable and Optimisation engine, OE and OF, include the appropriate PEP (Policy Enforcement Point) subcomponent for communication with Policy Server (PS). The possibility of including PEPs into OE and OF has been raised thanks to The Open Group's modification of the NGAC functional architecture, which unbundled the PEPs and Resource Access Points (RAPs) from the NGAC perimeter. As it states in D5.5, the appropriate PEP subcomponents for OE and OF are developed by following simple templates that include calls to the Policy Server through the RESTful Policy Query Interface. The PEP subcomponents consist of a single decision based on calling the Policy Server and either returning an error condition to its caller if the Policy Decision Point (PDP), a module of PS, returned deny or completing the access operation if the PDP returned *permit*.

The Policy Server API includes functions *initsession* and *endsession* which allow a session identifier to be registered as a proxy for a user identifier, where *user* denotes trusted components such as OE and OF. Then the *access* checks are made with the session identifier instead of a user identifier (i.e., its key). A particular SAFIRE component execution can be associated with a *user id* under a *session id*, which is a long string that is infeasible for an imposter to guess. The *session id* is passed to the OE (*user* according to the definition from D5.3) component to use when it makes requests to a policy enforcement point for OF (*object* according to the definition from D5.3) access. When the OE container instance is initiated by the SD component (as described earlier in this document), the initiator registers a *session id* to the policy administration API and passes that *session id* to the OE container. Only the initiator (SD), not the OE container, has the authorisation to call this API (enforced by its authorised TLS connection to the server). When the containers want to access their data, they make a request to a PEP for that data kind, supplying the *session id* in the request. The PEP, in turn, asks the PDP through the Policy Query Interface *access* request, using the *session id* instead of a *user id*, whether the access is permitted according to the policy and then enforces this decision accordingly. In this scheme, the PEPs own the data, that is, they are granted exclusive access to all the data so that they can enforce access according to the policy decisions of the PDP. The same scheme is followed when OE initiates the OF container.

The whole communication scheme based on SSF is presented in Figure 9. In the figure, two session ids are generated: *session_id* is generated by the SD module using the SD's user key (*sd_user_id*) and *session2_id* is generated by OE using the OE's user key (*oe_user_id*). The *access* call to PS checks whether the user (either SD or OE) is permitted to *execute* the object (OE and OF, respectively). If the access is permitted, the component functionality is executed and the results are returned to the invoker. Then the session is ended. In the figure, the optional PA module is not present. Its presence would require establishing the third session based on the OF's user key (*of_user_id*) and then using it for invoking the functionality of PA.

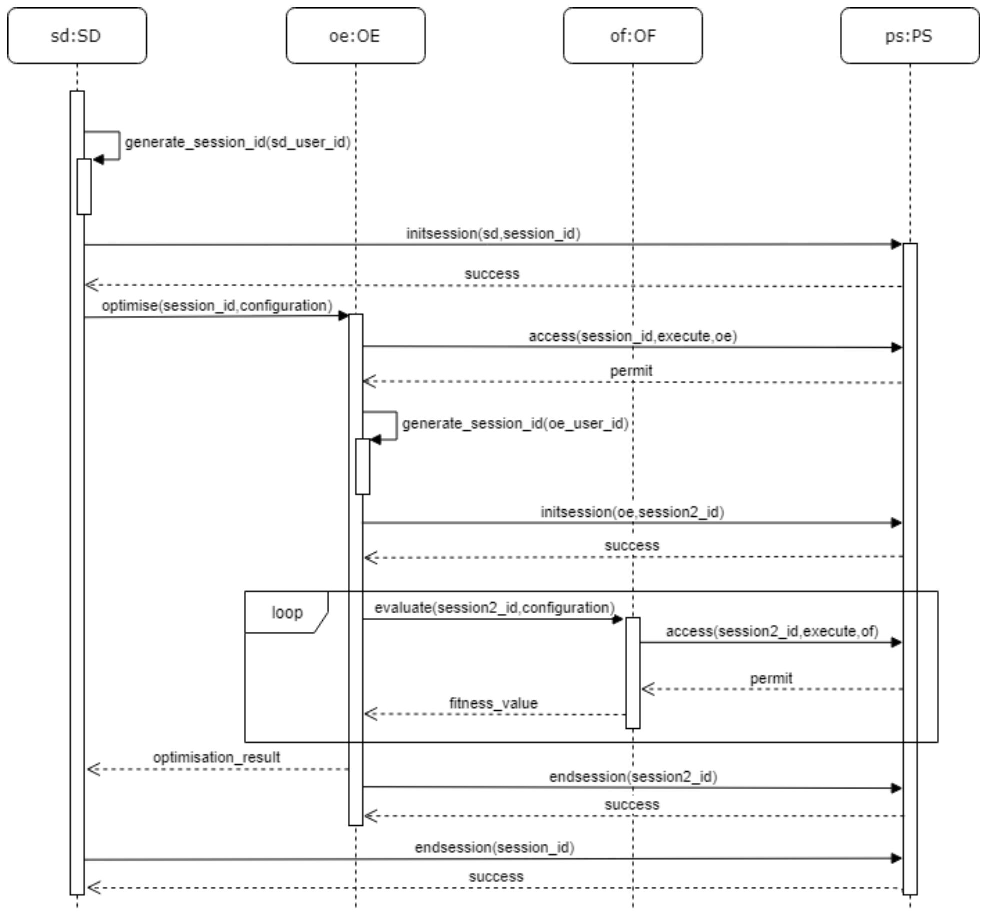

sd:SD
oe:OE
of:OF
ps:PS
generate_session_id(sd_user_id)
initsession(sd,session_id)
success
optimise(session_id,configuration)
access(session_id,execute,oe)
permit
generate_session_id(oe_user_id)
initsession(oe,session2_id)
success
loop
evaluate(session2_id,configuration)
access(session2_id,execute,of)
permit
fitness_value
optimisation_result
endsession(session2_id)
success
endsession(session_id)
success

Figure 9: Communications among components in OAS use case

6. Integration with Situation Determination

As referenced from deliverable D4.5 Final Specification of Situational Awareness Services, “The situational awareness services provide functionalities to handle for example configuration files, log-files. Furthermore, the situational awareness services provide abstract functionalities to access the Policy Server of the SAFIRE SPT framework. This abstract functionality can be extended by real implementations inside business case specific customisations. Thereby, all customisations built on top of the general situational awareness services are automatically NGAC aware.”

Situation Determination (SD) is connecting to other modules using TLS. Thereby, SD is not acting as server, it is connecting to other modules as client. It requests data from Data-Ingestion, requests data from Predictive Analytics and, calls the OE.

The components of the SD engine include the appropriate PEP (Policy Enforcement Point) subcomponent for communication with Policy Server (PS). The PEP subcomponents of the SD consist of a single decision based on calling the Policy Server and either returning an error condition to its caller if the Policy Decision Point (PDP), a module of PS, returned deny or completing the access operation if the PDP returned *permit*.

A policy file written in the declarative policy specification language has been created for SD for each Business Case. It indicates the resources to be protected and distributes the access rights to external parties. This file is loaded into the ngac server once during the setup phase of the SAFIRE solution. An exemplary policy file for the OAS Business Case can be found in Table 3

Table 3: OAS policy

```
policy('OAS_Policy','OAS Enterprise', [
                user('SD'),

                object('Mixer 1'),
                object('Mixer 2'),
                object('Mixer 3'),
                object('Mixer 4'),
                object('Mixer 5'),
                object('Mixer 6'),
                object('Mixer 7'),
                object('Mixer 8'),
                object('Mixer 9'),
                object('Mixer 10'),

                object_attribute('OAS Factory'),

                assign('Mixer 1','OAS Factory'),
                assign('Mixer 2','OAS Factory'),
                assign('Mixer 3','OAS Factory'),
                assign('Mixer 4','OAS Factory'),
                assign('Mixer 5','OAS Factory'),
                assign('Mixer 6','OAS Factory'),
                assign('Mixer 7','OAS Factory'),
                assign('Mixer 8','OAS Factory'),
                assign('Mixer 9','OAS Factory'),
                assign('Mixer 10','OAS Factory'),

                policy_class('OAS Ecosystem'),
                connector('PM'),
                assign('OAS Ecosystem','PM'),
                assign('OAS Factory','OAS Ecosystem'),

                associate('SD',[r],'OAS Factory')
]).
```

This file declares the resources to be protected, which are the 10 Mixers in OAS case. Each resource is being assigned to a parent element *OAS Factory*. The user of the resources will be the Situation Determination module, for which a user *SD* has been declared. The assignment of the read access right to the *OAS Factory* by *SD* has been declared in the statement *associate('SD',[r],'OAS Factory').* For the other two Business Cases there exist corresponding policy files with an analogue structure.

Within the Situation Determination module when resource access is being requested, a HTTP REST call to the ngac *Policy Query Interface* will be initiated. Find below an examplary http call requesting access to element *OAS Factory* by user element *SD:*

```
http://localhost/pqapi/access?user=SD&ar=r&object=OAS Factory
```

By this request, ngac will infer access to the 10 Mixers assigned to *OAS Factory* and return with *grant.* For authentication purposes, an a priori defined authentication token will be sent together with each access request to the ngac server.

Figure 10 shows an overview of the data flow from the production machines to data ingestion / Nifi layer involving the security layer:

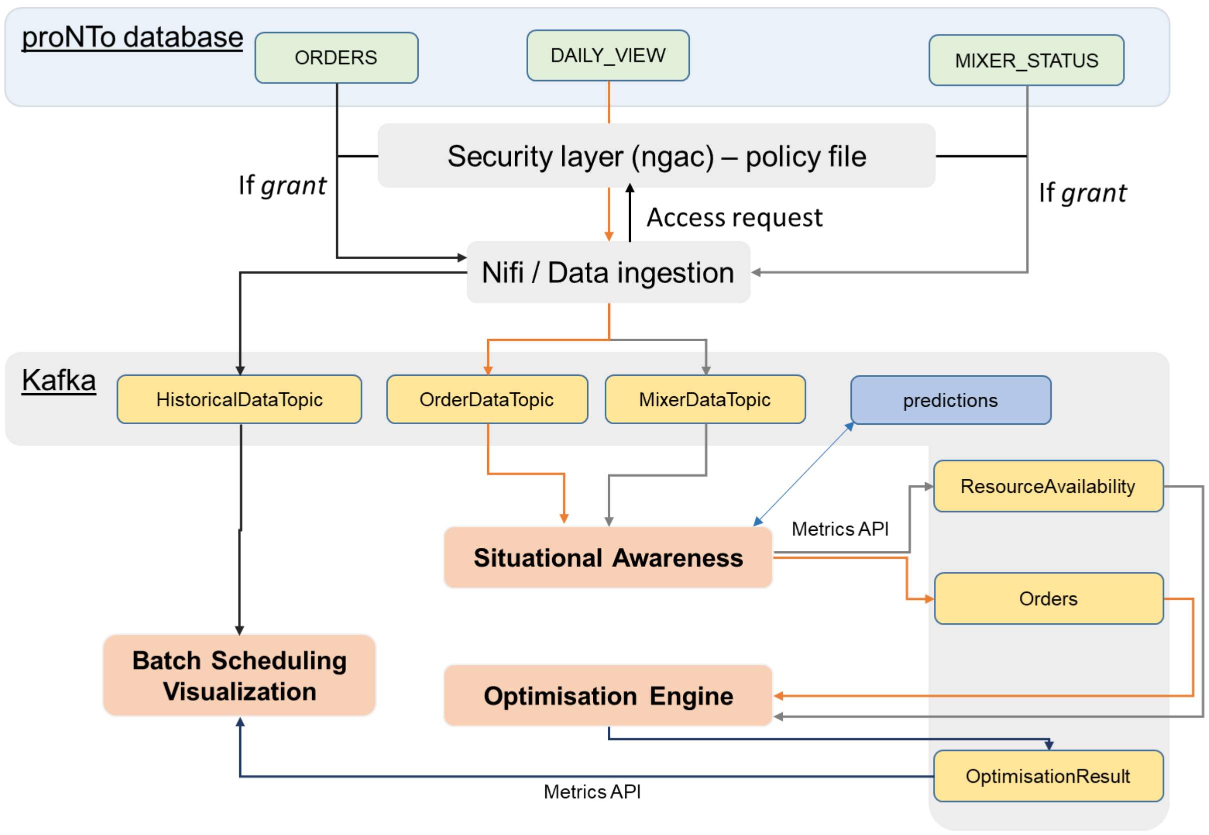


Figure 10 Data flow including security layer

This results in the following enhanced workflow:

- Data ingestion will continuously (i.e. an adjustable time interval, in OAS case in its current state 30 seconds) request data from the production machines using the Nifi interface
- Before the data is being processed, read access is being requested from the security layer, resulting in the following conditional flow
    - If ngac returns *grant*, then the data is being further processed into the Kafka topics as depicted in the figure above and made available for Situation determination services
    - If ngac returns *deny*, then there is no further processing of the data possible

On software side, this control mechanism is being realised within the *Kafka Manager*, which manages the processing of the data through the Kafka topics.

7. Installation and Operation

7.1. 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.

7.1.1. Install SWI-Prolog

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

7.1.2. 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.

7.1.3. 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> ‘

7.1.4. 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.

7.1.5. 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.

7.2. 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.

7.2.1. Install SWI-Prolog

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

7.2.2. 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.

7.2.3. Initiating the ‘ngac-server’

The ngac server may be started form the ‘ngac’ policy tool or it may be started with a compiled executable. This is preferable since it allows the command line options to be used.

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)." or "server(PortNumber,Token).", where PortNumber is an unused TCP port and Token is a symbol that the server will require when policy administration APIs are called. The server will be started and will be listening to that port for calls to its RESTful API. A server started without the second argument will expect the default admin token (the string "admin_token", without the quotes) in the policy administration API calls.

7.2.4. Test the installed 'ngac-server'

There are shell scripts of curl commands included with the source (servercurltest.sh and others) in the TEST subdirectory. These scripts can be run to send a sequence of requests to the server to test for known correct answers.

8. NGAC CUSTOMISATION

8.1. CUSTOMISATIONS AND EXTENSIONS TO NGAC FOR SAFIRE

There are customisations and extensions to the NGAC standard and reference implementations represented in our implementation. In some cases these have been done in a previous version of our implementation, and in some cases there are extensions to the NGAC Standard that were done for the SAFIRE project.

SAFIRE SPT Framework seeks to make advancements by enabling coherent system-wide security policy and enforcement in IIoT systems. To enable this, the SAFIRE Security Framework should provide an expressive, dynamic, and comprehensible security policy description and enforcement vehicle. The base requirements are met by the policy modelling and enforcement framework expressed in the NGAC Functional Architecture and related standards.

We intend to illustrate how the application of NGAC can provide a unifying approach to policy definition and access control in a dynamic FoF IIoT environment such as that which is the result of reconfiguration and optimisation as in SAFIRE. To do so we have made NGAC more usable, deployable in more diverse environments,, made it to be more easily extensible for new kinds of protected objects, and made it more supportive of large complex policies and policy composition than have been its past reference implementations.

We have described our own version of the NGAC functional architecture with “unbundled” PEP and RAP as shown in Figure 1. We have decoupled the components by giving them RESTful APIs. This architecture enables a more extensible implementation of NGAC by easing the addition of new protected object kinds. Another feature of our version of the architecture is a lightweight Policy Server that places many fewer demands on the operating environment and thus is more portable.

The extensions of our current implementation include:

- Extensions to the declarative policy language including declarations to enable future static and dynamic checks of policies;
- Support for policy composition;
- Unbundled PEPs and RAPs for new object classes;
- Constrained dynamic policy change – add and delete users, objects, user attributes, object attributes; add and delete assignments of users to user attributes and assignments of objects to object attributes;
- Full Policy Query and Policy Administration APIs;
- Addition of sessions to create a binding between a user identity and a successful user authentication that can be used instead of the user identity in an access query;

- Policy graph rendering command;

Declarative Policy Language

We have previously developed a simple declarative policy specification. Our expressed intention was to implement additional features in the language, including:

- *Prohibitions*, a construct to explicitly specify accesses that are to be denied (in addition to *default deny*), is a feature of the NGAC policy framework not previously implemented in our declarative language. Prohibitions have not yet been implemented as we have not found a need for them in the use cases that we have considered thus far.

- Ability to define new object classes – Currently, object classes are arbitrary identifiers that are not checked. This has been done.

- Ability to define specific operations that correspond to an object class – This will permit more precise type checking of object operations in policy rules. This has been done.

- Explicit support for policy composition in the language – Previously, policy combination was a command in the 'ngac' tool but not part of the language. The needed features are discussed more in the following section. Composition has been fully implemented in the policy server as well as in the policy tool. This has been provided as an alternative to the composition declaration in the language.

The definition of the declarative policy specification language given in APPENDIX A – has been extended to include the new language features in these extensions.

Policy Composition

For large heterogeneous IIoT systems there needs to be a representation in the policy language for the composition of policies and support for various forms of composition. The previous implementation of the 'ngac' policy tool did not fully support combined policies such as the example in the APPENDIX (Figure 13 and Figure 14). The implementation has now been extended to correctly handle such a simple examples. However, we anticipated a need to express policy composition *within the language* not just as a command outside of the language. We have extended the declarative language to include declarative policy composition and the implemented language processor accepts this extension. The processing to internally generate the declared composed policy has not been completed in the FP although much of the needed machinery is in place. However, the need has been met by full implementation of composition through the server policy administration API and in the policy tool command language.

Unbundled PEPs and RAPs for new Object Classes

Application developers need to be able to adapt to NGAC without waiting for further NGAC development before their needed resources can be available under NGAC authority. The developer already knows the resource and how it is to be accessed by the application. It should not be necessary to modify the core NGAC implementation for every such example (as was the case with the NGAC reference implementation). Instead, the developer should be able to construct all of the components of the object access path and test the application without involvement of the NGAC server if necessary. The necessary components are small trusted components that consist primarily of code that probably exists already in some form with the application. The developer must extract relatively small bits of code from the application and place them into templates for Policy Enforcement Points (PEP) and Resource Access Points (RAP). The Policy Enforcement Interfaces and Resource Access Interfaces are thus under the control of the developer.

We have developed examples and templates for PEPs and RAPs that application developers can use to more easily develop PEPs and RAPs, enabling them to add new kinds of protected resources.

8.2. IMPORTING POLICIES TO THE SERVER

A rich API has been added to the policy server for policy administration, including loading, unloading, and combining of policies; as well as runtime policy modifications.

8.2.1. Modifying Policy at Runtime

In addition to loading policies the server has been extended with ***add*** and ***delete*** APIs to modify loaded policies by adding or deleting individual policy elements and assignment relations. This is a limited form of dynamic policy modification that includes:

- Add user
- Add user attribute that is not part of an association (non-associated)
- Add assignment of a new or existing user or user attribute
- Add object
- Add non-associated object attribute
- Add assignment of new or existing object or object attribute
- Delete user and assignment of user to user attribute(s)
- Delete object and assignment of object to object attribute(s)
- Delete empty non-associated user attributes or object attributes

Note that this entails restrictions on additions and deletions. Attributes need to be empty to be deleted, and attributes that are part of an association cannot be deleted. Originally, there was no addition or deletion of associations but the implementation has now been extended to permit this.

Policy change is achieved through a sequence of calls to the ***add*** and ***delete*** APIs of the Policy Administration Interface. Appropriate checks of the restrictions are made when these calls are made to keep the policy structure in a consistent state. Assignments must be deleted before the entities assigned can be deleted. Conversely, entities must be added before assignments can be made.

8.2.2. Policy Composition

The SAFIRE implementation of the policy server has corrected previous deficiencies in policy composition by modifying the internal data structures and algorithms. This version also includes a new distinct form of composition of 'all' loaded policies.

8.2.3. Persistence of the Server Policy Database

Currently policies may be loaded into the server and the limited forms of changes noted above can be made. We considered implementation of persistence of the PIP but have not implemented persistence in the FP because of issues concerning the management of dynamic policies across multiple policy server execution sessions. We currently have a constrained form of policy change and it is not yet clear that starting from a previous (possibly uncertain) state, rather than a deterministically reproducible state, is better. By recording the sequence of changes to a known initial policy state (from a policy file) the state after changes can be reliably reproduced. We will reconsider persistence of the PIP if any of our use cases require it.

8.3. NGAC DEVELOPER FEATURES AND CUSTOMISATION

8.3.1. Customisation of the NGAC components

The 'ngac' policy tool presents a command line interface that offers a defined set of commands. The command interpreter also offers selective tracing of commands for development and debugging of the tool itself. The 'ngac' command interpreter is easily extensible for new commands, and this ability has been frequently used during the implementation of the 'ngac' software. The syntax and simple semantic checking of commands are achieved declaratively, and the addition of a new command is straightforward. There are two levels of commands: a restricted set for ordinary users (typically security administrators) and an expanded set that includes commands that are primarily of use to the tool developers. The expanded set includes (help is available in the tool for these):

- inspect(item) (extensible in command.pl to display any internal values)
- aoa(user) – show all object attributes for user in current policy
- demo(demo_name) – run a prepared demo
- los(policy) – show logical object system for specified policy
- reinit – reinitialize policy storage
- set(param,value) – display or set parameter value
- traceoff – turn off tracing
- traceon – turnon tracing
- traceone – trace a single NGAC command
- userlos(policy,user) – show the logical object system for user under policy

There are predefined command procedures ("procs") that exhibit some examples. At the ngac> prompt a predefined procedure (e.g. one may develop a command procedure 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 run with a pause before each command (useful for demos) with the command `proc(myproc,step)`.

It is instructive to read the file ***procs.pl*** that defines examples of predefined procedures. The procedures each consists of a sequence of the commands available at the ngac> prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

'ngac' commands can also be put into a file and executed as a script, without modifying the ***procs.pl*** file, using the `script` command, which like `proc` accepts the optional arguments `verbose` and `step`.

The 'ngac' tool has some self-tests built in. These may be run to ensure that everything is working correctly after installation, or after source code changes are made. The self-tests can be run by starting 'ngac' normally and entering at the 'ngac>' prompt the command "selftest."

The policy tool and policy server can be adapted in the following 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 for all the NGAC components are collected in one place and can be redefined.
- New HTTP callable server functions can be added.

8.3.2. Predefined command procedures

There are predefined command procedures ("procs") that run some examples. At the ngac> prompt a predefined procedure (e.g. one may develop a command procedure 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 run with a pause before each command (useful for demos) with the command `proc(myproc,step)`.

It is instructive to read the file ***procs.pl*** that defines the predefined procedures. The procedures each consists of a few of the commands available at the ngac> prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

'ngac' commands can also be put into a file and executed as a script, without modifying the ***procs.pl*** file, using the `script` command, which like `proc` accepts the optional arguments `verbose` and `step`.

8.3.3. Extension of the built-in tests

The 'ngac' tool has some self-tests built in. These may be run to ensure that everything is working correctly after installation, or after source code changes are made. The self-tests can be run by starting 'ngac' normally and entering at the 'ngac>' prompt the command "selftest."

The file TEST/spld_test.pl defines self test cases with expected results for the policy language processing and decision functions in the code file `spld.pl`. These tests use the test framework implemented by the file `test.pl`. Test for other modules may be created in a similar fashion, by creating a `TEST/mod_test.pl` file for a corresponding module `mod.pl` which must be modified to have an include directive for its test file.

8.3.4. Global parameters

Global parameters are set in the file `param.pl`. There are many values defined here that affect the operation of the NGAC components. Settable parameters (those that can be changed from the 'ngac' command line with the `set` command or internally from Prolog code with the `setparam` predicate) are itemized in a list `settable_params`. Adding new settable parameters requires the new parameter name to be added to this list and to the `dynamic` directive above it in a fashion similar to the other entries.

8.3.5. New policy server APIs

Using the existing APIs as an example, additional APIs may be added to the policy server by modifying the file `server.pl`.

9. SOFTWARE TOOLS

The implementation of the ‘ngac’ policy tool and the lightweight ‘ngac-server’ are intended to be simple and portable, with minimal external dependencies. Motivation for its development and the specific objectives are described elsewhere. The NGAG reference implementations have been very heavyweight with many external dependencies on languages, libraries and tools.

Our NGAC software is implemented in the Prolog language, which is well suited to representation and computation over access control policies in the NGAC framework. The Prolog implementation we use, SWI-Prolog, includes a visual editor and graphical tracer and a built-in make facility for rapid iterative development.

The only tools necessary for this implementation are SWI-Prolog and the libraries included in its release; we are currently using version 7.6.4. The release includes a version of Emacs with a Prolog display profile, though an editor of the developer’s choosing may be configured for invocation instead when editing Prolog source files.

10. Conclusions and Plans

We have succeeded in implementing a version of the NGAC standard that computes the policy calculations for an arbitrary policy or combinations of policies. Existing examples of NGAC policies, and examples added to exhibit newly implemented features, are used as built-in self-tests and regression tests to confirm that the tool computes the known answers and that changes made during development do not cause the implementation to regress from previously achieved correct operation.

Several extensions to the EP have been developed to create the FP. These include those capabilities needed to fulfil the contemplated application as a mechanism for endpoint access control in the present use cases.

We intend to continue to develop and enhance our NGAC implementation to meet new requirements that we encounter as we pursue our exploitation plans.

As has been demonstrated in our mapping of the IISF functional building blocks to the NGAC approach, there are numerous future potential options for using NGAC in multiple functional roles within deployments of a SAFIRE-enabled FoF system. When an NGAC-based approach seems that it would be beneficial, adding features to the present implementation and already planned extensions will be undertaken to further expand the capabilities and applicability of the implementation.

11. REFERENCES

[FGJ15]	David Ferraiolo, Serban Gavrila, and Wayne Jansen. Policy Machine: Features, Ar- chitecture, and Specification. National Institute of Standards and Technology, October 2015. NIST Internal Report 7987 Revision 1.
[G$^+$]	Gavrila et al. Policy machine source. https://github.com/PM-Master.
[Gav07]	Serban I. Gavrila. The Policy Machine: User Guide, January 2007.
[Int15]	InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control—Generic Operations & Abstract Data Structures, May 2015. INCITS Project CS1/2195-D, NGAC-GOADS, Revision 1.60 in review, to appear.
[Int16a]	InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control—Functional Architecture, February 2016. INCITS Project CS1/2194-D, NGAC-FA, Revision 0.70 in review, to appear.
[Int16b]	InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control— Implementation Requirements, Protocols and API Definitions, February 2016. INCITS Project CS1/2193-D, NGAC-IRPAD, Revision 0.20 in review, to appear.
[Int13]	InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control—Functional Architecture, 499-2013. INCITS Project CS1/2194-D, NGAC-FA, March 2013.
[Int18]	InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control—Functional Architecture, 499-2018. INCITS Project CS1/2194-D, NGAC-FA, Supersedes INCITS 499-2013, January 2018.
[SAF D5.2]	SAFIRE Project, D5.2 Early Specification of Security, Privacy and Trust Framework, Version 1.0, March 2018.
[SAF D5.5]	SAFIRE Project, D5.5 Final Specification of Security, Privacy and Trust Framework, Version 1.1, 14 November 2018.

12. APPENDIX A – NEXT GENERATION ACCESS CONTROL

12.1. NGAC OVERVIEW

The access control policy specification representations to be described are based on the framework provided by the Next Generation Access Control (NGAC) standard:

- Next Generation Access Control Functional Architecture (NGAC-FA) [Int16a]
- Next Generation Access Control Generic Operations and Data Structures (NGAC-GOADS) [Int15], and
- Next Generation Access Control Implementation Requirements, Protocols and API Definitions (NGAC-IRPAD) [Int16b].

Where the standards leave off the “Policy Machine” reference implementation is consulted for details.

We begin with a description of the access control framework. Following this, we describe the representations found in the standards or in the reference implementation, including a graphical representation, the low-level textual representation, and the low-level imperative command representation are distilled from the standards and the reference implementation. The low-level imperative command representation of policy is that of the reference implementation described in [Gav07]. It is capable of dynamic policy changes that can be done under the control of scripts triggered by events.

Also presented are some alternative policy representations that we have developed, including an alternative to the low-level imperative command representation and a novel declarative representation used by our ‘ngac’ policy tool. The declarative representation is cleaner and more intuitive. We have not yet investigated its extension for dynamic policies though our tool does have scripting capabilities that may be extended and adapted for the purpose.

Examples and figures following are taken from the NGAC standard document and from documents describing the Policy Machine [FGJ15, Gav07] reference implementation [G^+] of the NGAC standard. The examples are recast in the alternative declarative representation.

12.2. NGAC ROLE IN SAFIRE ARCHITECTURE

The IISF highlights the foundational role of security model and policy in its functional viewpoint. Critical aspects of this role are those addressed by NGAC.

NGAC is a novel approach to access control that affords unprecedented flexibility and the ability to represent and enforce arbitrary attribute-based access control policies within a unified framework. NGAC has not to our knowledge been applied previously in industrial manufacturing or any Industrial Internet of Things (IIoT) environment.

NGAC extensions will also be needed to address SAFIRE real-time data requirements for security policy enforcement.

The features and particulars of the NGAC reference implementation, and the published examples, suggest that it has thus far been applied to enterprise IT. However, the scalability and flexibility of NGAC make it a potent tool for addressing the scale, complexities, and security challenges of combined IT and OT as found in IIoT systems.

NGAC provides a framework and mechanisms that have the potential to unify the system-wide access control policies, and provide the ability to understand the net effect of the composed policies of the underlying mechanisms.

There are several extant reference implementations (RI) of NGAC, mostly under the name “Policy Machine” (PM), that have been developed in recent years by the principal authors of the NGAC standards. It has not been an apparent goal of these efforts to provide an industrially deployable implementation that supports heterogeneous environments, or that is readily extensible to new kinds of protected objects. Rather, the re have all been proofs-of-concept, all the more so with the latest versions. Nonetheless, with versions subsequent to the initial PM, the developers seem to have taken into consideration some of our comments and requests made over the past two to three years, with improvements such as independence from Microsoft Windows Server, independence from Windows Active Directory, and independence from LDAP. They have moved to store the policy information first in MySQL and then, as an option, in Neo4j[13], which is better suited to policies in the NGAC framework. As more recent prototypes have emerged, they seem to have focused on narrower aspects of NGAC functionality, rather than a complete system, such as improved algorithms, a Web-service-based NGAC server, RESTful APIs, and deployment in Docker containers, though, as far as we understand, not all backward-compatible with their more complete early PM versions. We continue to monitor the releases of their experiments, as it is clear that many of their recent developments are relevant to our concerns.

NGAC is described in the source documents [FGJ15], [Int15], [Int16a], [Int16b] and [Int18]. Several versions of reference implementations of NGAC are described in [G$^+$] and [Gav07].

The Open Group has implemented our own NGAC-related tools and a simple declarative language to express policies that comply with the NGAC framework. Specifically, a desktop command-line tool called ‘ngac’ that 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’?”.

The declarative language is easily read from a graphical representation of the policy, and is more intuitive than the imperative language of the PM RI. The ‘ngac’ policy tool can generate a translation of the declarative policy in the imperative language for import into the earlier PM servers. The present declarative language does not support the entire NGAC policy framework, currently lacking prohibitions and obligations. The present

[13] Neo4j is a graph database that is available in community and commercial versions.

specification of the declarative language is given in the text of this document. We anticipate making future extensions to this implementation.

12.3. NGAC MOTIVATION

According to the NGAC-FA Standard [Int16a]:

> *Next Generation Access Control (NGAC) is reinvention of traditional access control into a form that suits the needs of modern, distributed, interconnected enterprise. The NGAC framework is designed to be scalable, to support a wide range of access control policies, to enforce different types of policies simultaneously, to provide access control services for different types of resources, and remain manageable in the face of change.*

12.4. NGAC POLICY FRAMEWORK

The core constructs of the access control framework are:

- A set of basic elements – representing entities
- A set of containers of different types – to represent characteristics of basic elements
- A set of relations – to represent relationships among basic elements and containers

There is also a set of commands for the creation, deletion and maintenance of basic elements, containers and relations.

The basic elements comprise:

- Users – unique entities that are either humans, trusted programs, or devices
- Processes – system entities that have a reliable user identity and operate in a distinct memory
- Objects – resources to which access is controlled, e.g. files, messages, database records, etc.
- Operations – denote actions performed on elements of policy (either external protected resources or internal resources)
- access rights – enable actions to be performed on elements of policy (either external protected resources or internal resources)

Containers comprise:

- User attributes – defines membership on the basis of an abstract user capability or property. The members of a user attribute may be users or other user attributes. Membership is transitive.
- Object attributes – defines membership on the basis of an abstract object characteristic or property. Members of an object attribute may be objects or other object attributes. Membership is transitive.
- Policy classes – defines membership related to an access control policy, such as RBAC, MLS. Members of a policy class may be users, user attributes, object, or object attributes. Multiple policy classes may exist simultaneously.

Every user attribute, object attribute, and policy class has a unique identifier. Policies are expressed as configurations of relations of the following four types:

- Assignment – defines membership within containers, involves a pair of policy elements
- Association – defines authorized modes of access, it is a 3-tuple < *userattribute, accessrightset, attribute* >
- Prohibition – specifies a privilege exception, it is a 4-tuple (of 3 different kinds described below)
- Obligation – dynamically alters access state, triggered by an event; it is a 3-tuple < *user, eventpattern, eventresponse* >

The prohibition relation has three forms:

- <*user, accessrightset, inclusiveattributeset, exclusiveattributeset*>
- <*userattribute, accessrightset, inclusiveattributeset, exclusiveattributeset*>
- <*process, accessrightset, inclusiveattributeset, exclusiveattributeset*>

Events that trigger obligations may include the following information:

- Operation;
- User ID;
- Process ID;
- One of the user attributes of the process performing the operation;
- Policy element ids representing a resource or policy information; or
- One or more attributes of the resource or policy information on which the operation has been performed.

From the four configured relation types above, four types of derived relations can be computed for the purpose of making access control decisions:

- Access control entry – derived from association; < user, accessright >
- Capability – derived from association; < accessright, policyelement >
- Privilege – derived from association; < user, accessright, policyelement >
- Restriction – derived from conjunctive and disjunctive prohibition relations of the same form; restricts process from performing an operation against a policy element, based on process attributes.

13. APPENDIX B – NGAC-BASED SECURITY POLICY REPRESENTATIONS

There are several representations of policies based on the NGAC policy framework.

13.1. DECLARATIVE POLICY LANGUAGE REPRESENTATION

The Declarative Policy Language (DPL) is the sole vehicle for expressing NGAC policies in a form usable to the TOG-NGAC implementation. The syntax and semantics of the DPL are described in Section 2.1 of this document and the underlying policy framework in Section 12.4 of Appendix A.

13.2. GRAPH REPRESENTATION

Policies expressed in this framework are best considered as mathematical graphs. This is a very natural way to represent and to develop a policy. The examples presented here are represented as directed graphs in a particular layout that is conducive to interpreting the policy. Such graphs may be sketched manually or with a drawing utility as are the following examples. A `policy_graph` command has been added to the ‘ngac’ policy tool to create a graphical rendering of a loaded policy, as shown examples at the end of this section.

Figure 11 illustrates two examples of assignment and association relations as graphs. Figure 11(a) is an access control policy configuration with policy class “Project Access”, and Figure 11(b) is a data service configuration with “File Management” as its policy class. On the left side of each graph are users and user attributes, and on the right side are objects and object attributes. Arrows represent assignment relations and dashed lines represent associations. An association may also be thought of as a pair of assignments, the first an assignment of a user attribute to an operation set, and the second an assignment of the operation set to an object attribute. In the association **Division–{r}–Projects**, the policy elements referenced by **Projects** are objects **o1** and **o2**, meaning that users **u1** and **u2** can read objects **o1** and **o2**.

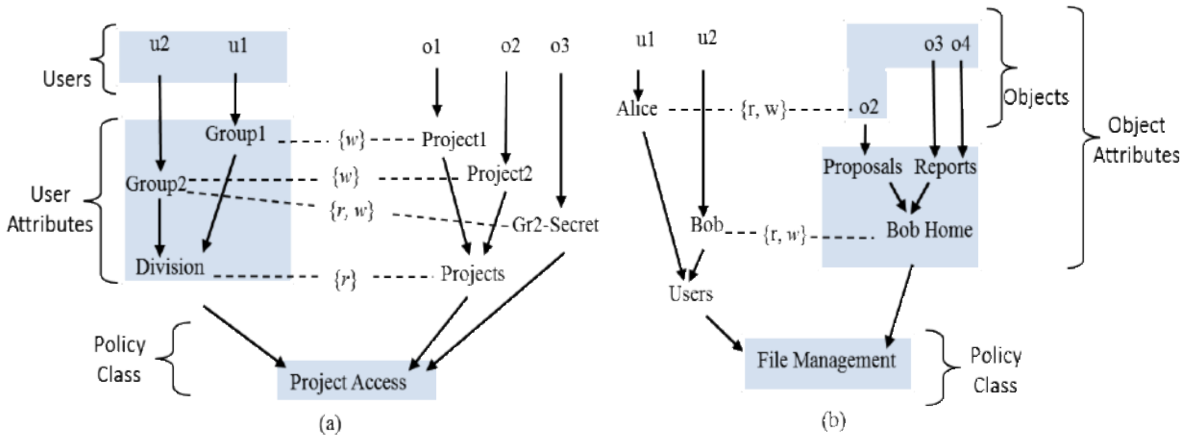


Figure 11: Assignment/Association Graphs

Figure 12 illustrates the independent derived privileges of the separate graphs in Figure 11.

(u1, r, o1), (u1, w, o1), (u1, r, o2), (u2, r, o1), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3)	(u1, r, o2), (u1, w, o2), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3), (u2, r, o4), (u2, w, o4)

Figure 12: Independent derived privileges from Figure 11(a) and (b)

The policy specification technique allows for complex policies to be built up from separate policies or policy fragments. For example the two policies of Figure 11 when combined yield the policy graph illustrated in Figure 13. The derived privileges of the combined graph is shown in Figure 14.

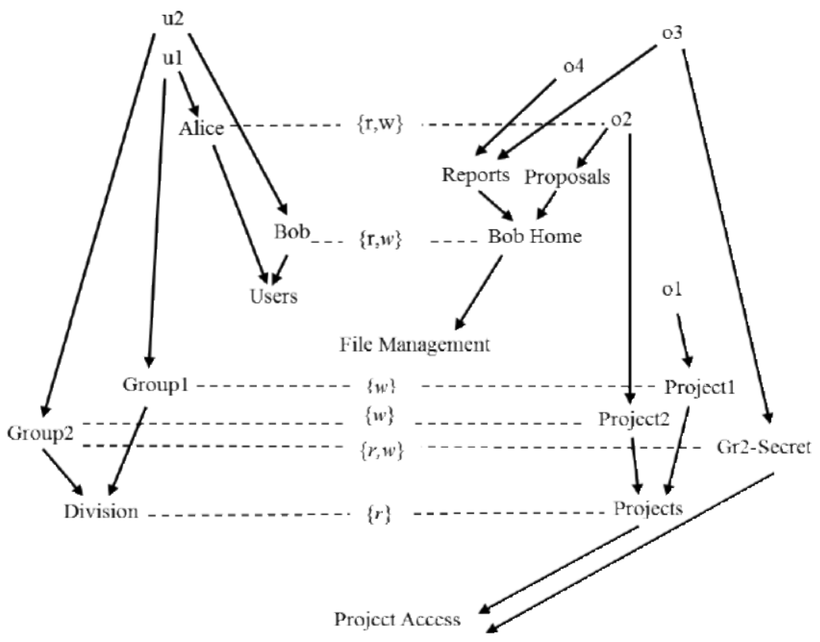


Figure 13: Combined policy graphs of Figure 11

(u1, r, o1), (u1, w, o1), (u1, r, o2), (u2, r, o1), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3), (u2, r, o4), (u2, w, o4)

Figure 14: Derived privileges of the combined graphs of Figure 11

Figure 15 and Figure 16 respectively show the output generated by the ‘ngac’ tool’s `policy_graph` command for the policies (a) and (b) introduced in Figure 11.

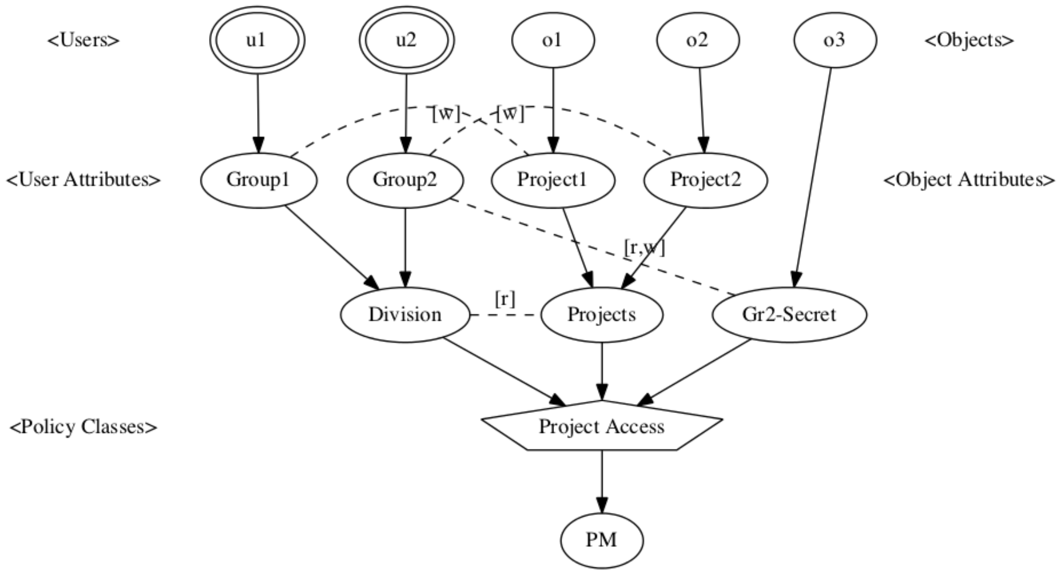


Figure 15: 'ngac' policy_graph rendering of policy (a)

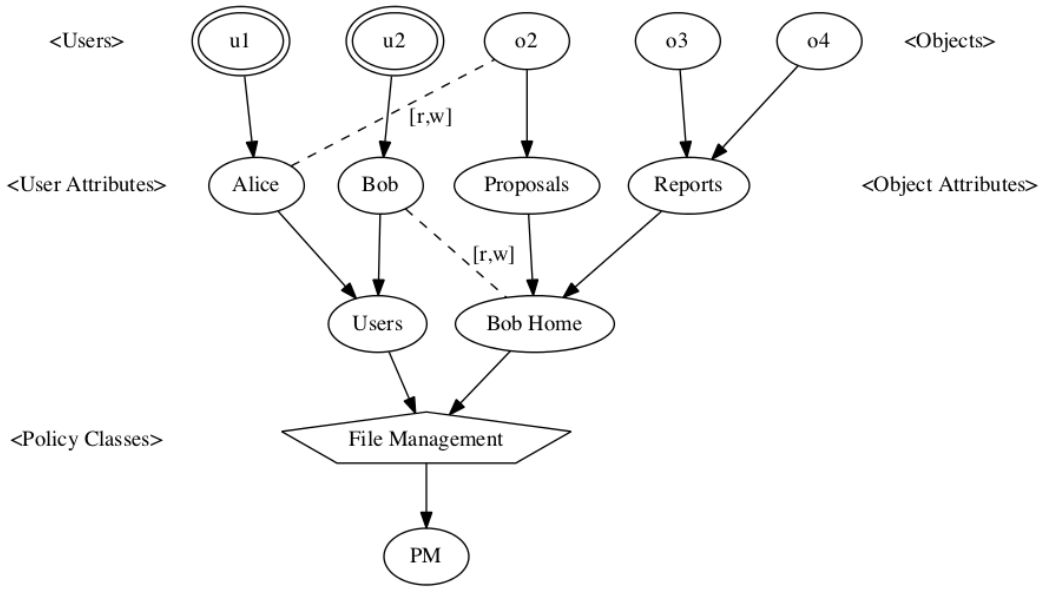


Figure 16: 'ngac' policy_graph rendering of policy (b)

13.3. LOW-LEVEL REPRESENTATIONS

Earlier PM reference implementations used two low-level representations, a low-level textual representation of the policy machine graph, and a low-level imperative command representation that represents primitive actions that the PM Server can perform to build and manipulate elements of its internal policy model. The Admin Tool in the PM reference implementation directly manipulates the system-wide model maintained in the PM Server by passing it commands in the imperative language.

Live update of the system-wide model can easily disrupt normal operations, and is not a viable practice for policy development and testing. We implemented the original standalone policy tool called ‘ngac’ for this purpose. The low-level imperative representation could be generated by our ‘ngac’ policy tool, creating files that could be imported by the PM Server. These capabilities are no longer relevant since our ‘ngac-server’ implementation has the ability to load policies represented in our Declarative Policy Language and has a Policy Administration Interface that enables policy manipulation in the server.

14. APPENDIX C – NGAC POLICY DEVELOPMENT

We discuss the process of developing policies to use with the NGAC Policy Server and present the ONA policy as an example. The reader should be familiar with the NGAC policy framework described in Section 12.4 Appendix A and the policy representations described in Section 2.1 and Section 13 Appendix B.

14.1. METHODOLOGY FOR DEFINITION OF POLICY ELEMENTS

We begin with an overview of the methodology for developing an NGAC policy.

1) Identify the distinct objects and object types to be protected (protected resources).

2) Identify the controlled operations on each object or object type.

3) Identify the distinct users and groups of users that are linked to identities that are reliably available at run time.

4) Identify, for all the users and objects, a collection of attributes that can be used to characterize users or objects. These should be *binary attributes*, that is, each is an attribute that a user or object either has or does not have. Attributes can be thought of as a set to which a user or object may belong.

5) Make assignments of each user to appropriate user attributes, each user attribute to other user attributes, objects to object attributes, and object attributes to other object attributes.

6) Connect the resulting graph 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'. Multiple policy classes, and their descendant attributes, may belong to the connector.

Some of the items introduced above are now described further.

14.1.1. Attributes

Often it is convenient to create attributes that make sense for the domain whether or not they correspond to actual runtime entities. The organization for which the policy is being developed should identify user attributes and object attributes independently of particular policy details. The set of basic attributes for the organization should apply to most policies. Additional attributes may be needed to provide distinctions that are needed for particular policies, but the set of basic attributes would still be valid.

Things that should be considered with defining the attributes appropriate for the policies of an organization would include:

- Job titles
- Hierarchical management structure and roles
- Functional roles

- Non-employee roles
- Customer roles
- Departmental and project structures
- Physical facility divisions
- Organization of authority over ("ownership of") resources
- Logical or physical organization of resources
- Kinds of resources/data
- etc.

Attributes pertaining to users and groupings of users should reflect the logical structure of relationships; the same for objects and groupings of objects.

For example, let us create a policy to convey the simple idea of privileged access to certain objects. Suppose we have a set of users some of whom, administrative users, have access to objects that other users do not. Any user can be designated as an ordinary_user or as an administrative_user, and data objects designated as an unrestricted_object or arestricted_object. The user attributes are all_users, ordinary_user, and administrative_user; the object attributes are all_objects, restricted_object, and unrestricted object. In Figure 17 we see a policy graph containing this core set of attributes and their relations (blue arrows). We can then assign individual users and objects to these categories (red arrows). Finally we can represent the intended access associations among user attributes and object attributes (green arrows).

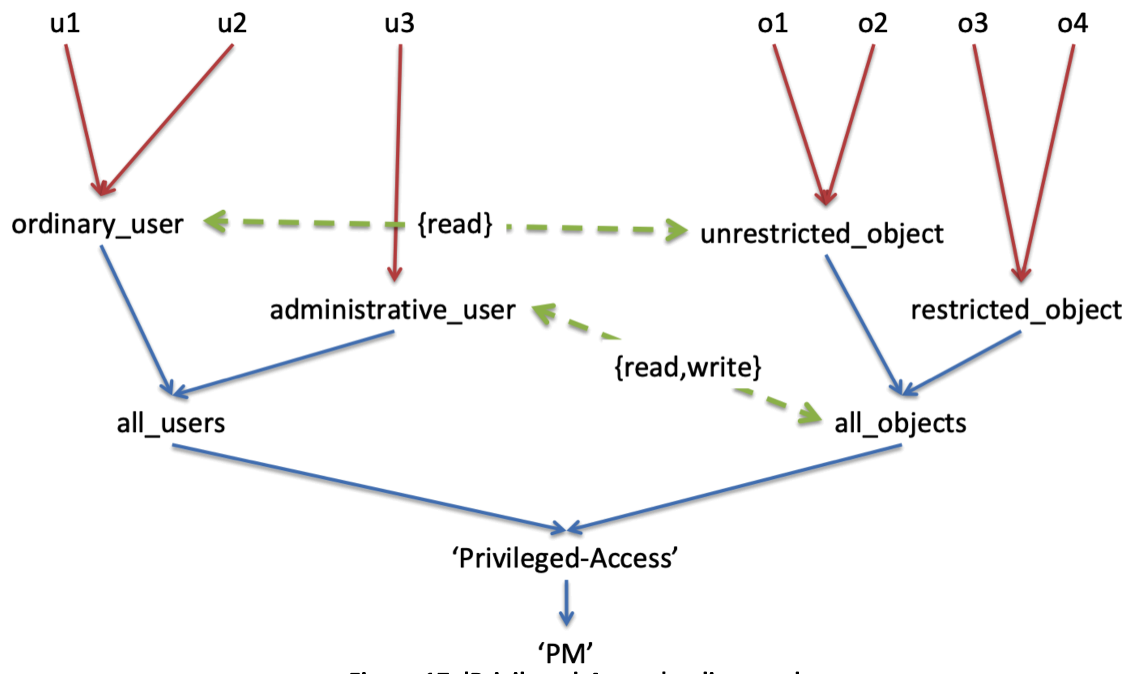
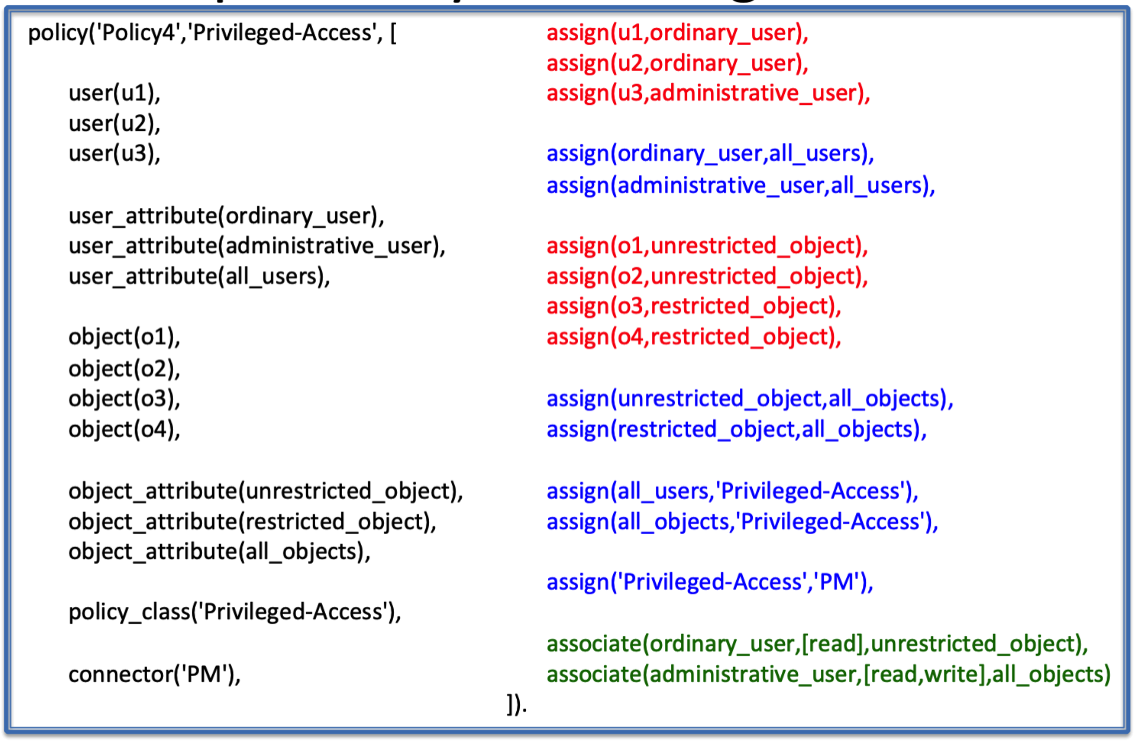


Figure 17: 'Privileged-Access' policy graph

The declarative specification of the policy is shown in Figure 18 with the rules of the specification coded in the same colors as the corresponding elements of the graph. This specification embodies the policy that administrative users are allowed to read and write all objects (both restricted and unrestricted), while ordinary users may only read unrestricted objects. It can easily be seen that changes to the organizational policy, such as permitting ordinary users to both write and read unrestricted objects, does not require any change to the defined attributes or their logical structure. Only the association between the ordinary_user attribute and the unrestricted_object attribute must have an added access right for write. If the organizational policy was amended to permit ordinary users to read, but not to write, restricted objects, then again the attributes would not change but a new association would be made between the ordinary_user attribute and the restricted_object attribute with the read access right only.



```
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)
]).
```

Figure 18: Declarative specification of the 'Privileged-Access' policy

14.1.2. Web services as policy objects

For web services as objects the methodology above may be modified slightly. One still identifies the distinct objects (or resources) and the operations permitted on them. Often there are multiple possible operations on the same object. These operations will be performed by a Web service, either a URI per object type, a URI per object, or a URI per operation per object, as determined by the definition of the service associated with each URI and its arguments.

How the services are defined all depends on how one wants to organize the concepts in the policy and whether every API (URI) is to be thought of 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 may be defined 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.

As an example, suppose the contents of a file A are to be provided as a web service. There are multiple ways to package this service. One way is to provide separate read and write operations within the file A object's service:

```
fileAservice read
fileAservice write
```

which could be made the Web APIs

```
fileAserviceRead
fileAserviceWrite
```

This would be reflected in the policy as:

```
policy( myFileServicePolicy, 'FSP'  [

        user(u1),
        object(fileA),
        object_attribute(served_file),
        assign(fileA, served_file),
        associate(ordinary_user,[read],served_file),
        …]).
```

policy queries would look like access <u1,read,fileA>, where the operations are *read* and *write* and the service is *file access*.

Alternatively:

```
policy(  myFileOpServicePolicy, 'FOSP', [

        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 against this policy would look like access <u1, call, fileAread>, where the operation requested is *call* and the service provided is *read from file A*.

14.2. EXAMPLE – DEVELOPMENT OF AN NGAC POLICY FOR ONA

This section presents an example of applying the methodology to the development of an NGAC policy for an ONA use case. First, working with the ONA partner we gathered information about the security environment, and identified security objectives and parameters of the ONA environment.

14.2.1. The ONA security environment

ONA identified its overall security objectives:

1) To identify and authenticate users
2) Protection and the confidentiality of the information.
3) Users account Management integrated in third party platforms.
4) The right information [access] for every user according to his profile or role.
5) Capability for registering Security related activity issues (access control, backup/restore operations, configuration changes, forensic auditing, …
6) System integrity. To prevent unauthorized manipulations of the system (software execution, software install, communication, update/upgrade management, …).

ONA identified its legitimate ***actors/users of Data Assets***:

1) ONA (OEM). The machine manufacturer.
2) ONA's customer, usually the machine owner and the company operating the machine. In the future this scenario could be more complex: machine renting, owner that doesn't operate the machine, …
3) Third party companies giving services based on machine data: technical service providers (maintenance, calibration, equipment certification, …), components and spare part providers, financial companies (insurance, exporting/trading, …), … even the government/public sector (energy [provider …])

ONA identified the ***Data Assets needing protection***, ***threats to the assets***, and organizational ***policies of protection*** for each actor/user.

For OEM (ONA) itself and its customers, the ***Assets***:

- Manufacturing data (programs, scripts, reference points, set-up data, technology files, etc.)
- Machine manufacturing data (calibration, axis compensation, etc.)
- Software systems (CNC/PLC software, firmware, etc.) (Technically a machine CNC software update/upgrade can be a re-configuration process in SAFIRE, so this software can be seen as a "data asset".

ONA identified the ***threat***s to these assets:

- Unauthorized manipulation can generate equipment malfunction affecting the OEM trade image and reputation,
- Unsatisfied customers
- Legal issues if applied (turn key project with defined technical targets and compromises, etc.)

The ***policy of protection*** for these Assets:

- Only OEM authorized actors: application/service engineers, R&D staff, dealers, etc.
- Actors permitted / not permitted to act on an Asset, permissions may be different per actor

For OEM’s customers and Third party companies, the ***Assets***:

- Machine usage data
- Customer behaviour data
- Other types of aggregated data

The ***threat***s to these assets:

- Confidential data (people of companies)
- Intellectual property rights associated to manufacturing processes, etc.
- Other types of rights that must be preserved because of legal regulations

The ***policy of protection*** for these assets:

- Only authorized actors can access the data asset.
- OEM and/or any other company providing cloud SAFIRE services can be excluded from the authorized actors list depending on legal contracts
- This is a key topic under research. Similar problem with the assignment of aggregated data.

Finally, Assumptions that may be made about the environment:

- Protections are provided against Physical manipulations of machines (covers, cabling, devices, etc.)
- Protection of data environment in private spaces (factory, workshop, etc.)
- Personnel data (face images, fingerprint, etc.) (in custody of dedicated systems?)

14.2.2. ONA security problem

From the information gathered about the security environment we present a succinct statement of the security problem:

Assets to be protected include:

- Manufacturing data
- Machine configuration data
- Machine usage data
- Customer behaviour data

Threats to the Assets (reiterated more succinctly):

- Unauthorized modification leading to equipment malfunction
- Exposure of confidential data
- Compromise of IPR
- Violation of rights guaranteed by Regulations

Policies to be followed:

- Only authorized access – default is no access
- SAFIRE services in Cloud are subject to access controls

Assumptions on the Environment:

- Adequate physical protection is in place – tampering is covered
- Physical spaces where data processing systems and manufacturing machines reside are secured – untrustworthy personnel is covered
- Personnel data are protected – by segregation from other data on separate systems, and thus are covered

14.2.3. ONA policy formalization

We now define the elements of an NGAC policy that will cover aspects of the security problem that are suitable for an NGAC solution. Without identifying specific users or data objects we can identify attributes of both that are meaningful to the organization and are likely to provide a framework for expressing access control policy.

NGAC user attributes:

- 'ONA' – the company
- 'ONA Staff' – staff of the company
- 'ONA Mgt' – company management staff
- 'ONA FEng' – company field engineering staff
- 'Cust' – staff of all ONA customers
- 'Cust A' – staff of a particular ONA customer A
- 'Cust B' – staff of another ONA customer B
- '3rd Party' – staff of all third party companies
- '3rd Party C' – staff of a third party company C

The user attributes defined above have natural hierarchical relationships that are captured in the assignment of user attributes to other user attributes. Graphically, an attribute that is encompassed by another attribute (contained in) is assigned to the encompassing attribute and is presented above the encompassing attribute. We say that the assigned attribute is ***an ascendant of*** the attribute to which it is assigned. The user attributes defined above, along with the assignments implied by their intended interpretation, are presented graphically in Figure 19. This figure was generated by applying the `policy_graph` 'ngac' command to the user sub-graph policy fragment presented in Table 4 as a DCL specification.

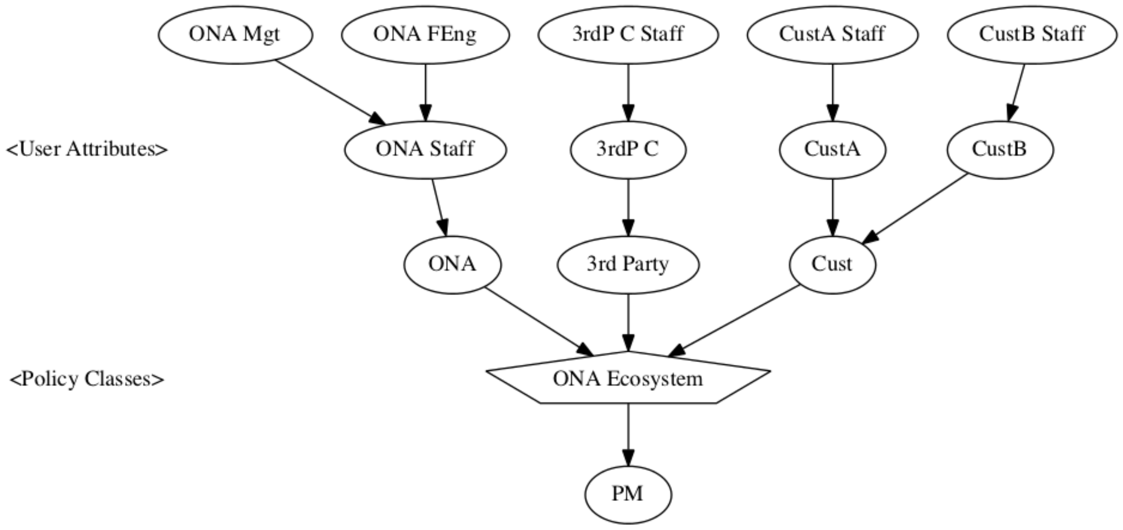


Figure 19: Representative user attribute sub-graph of the ONA Ecosystem policy

Table 4: ONA user attribute sub-graph policy specification

```
    policy('ONA_Policy_U','ONA Ecosystem', [
                         user_attribute('ONA'),
                         user_attribute('ONA Mgt'),
                         user_attribute('ONA FEng'),
                         user_attribute('ONA Staff'),
                         user_attribute('Cust'),
                         user_attribute('CustA'),
                         user_attribute('CustB'),
                         user_attribute('CustA Staff'),
                         user_attribute('CustB Staff'),
                         user_attribute('3rd Party'),
                         user_attribute('3rdP C'),
                         user_attribute('3rdP C Staff'),

                         assign('ONA Mgt','ONA Staff'),
                         assign('ONA FEng','ONA Staff'),
                         assign('ONA Staff','ONA'),
                         assign('3rdP C Staff','3rdP C'),
                         assign('3rdP C','3rd Party'),
                         assign('CustA Staff','CustA'),
                         assign('CustB Staff','CustB'),
                         assign('CustA','Cust'),
                         assign('CustB','Cust'),
                         assign('ONA','ONA Ecosystem'),
                         assign('Cust','ONA Ecosystem'),
                         assign('3rd Party','ONA Ecosystem'),

                         policy_class('ONA Ecosystem'),
                         assign('ONA Ecosystem','PM'),
                              connector('PM')
    ]).
```

NGAC object ***attributes*** (these are not the objects themselves):

- 'Cust Behav' – contains behaviour data for operator of machine
- 'Mach Usage' – contains machine usage data
- 'Mach M-Data' – contains machine manufacturing data
- 'Mach C-Data' – contains machine configuration data
- 'MachX M-Data' – contains manufacturing data for ONA machine X
- 'Owner Data' – contains data the owner may see
- 'All Data' – self explanatory

NGAC "objects" (data assets):

- 'MachA1 Cust Behav' – the customer behaviour data for machine A1
- 'MachA1 Usage' – the machine usage data for machine A1
- 'MachA1 Calib' – the calibration data for machine A1
- 'MachA1 Axis' – the axis compensation data for machine A1
- 'MachB1 Cust Behav' – the customer behaviour data for machine B1
- 'MachB1 Usage' – the machine usage data for machine B1
- 'MachB1 Calib' – the calibration data for machine B1
- 'MachB1 Axis' – the axis compensation data for machine B1

The object attributes and the objects defined above, along with the assignments implied by their intended interpretation, are presented graphically in Figure 20. This figure was generated by applying the `policy_graph` 'ngac' command to the user sub-graph policy fragment presented in Table 5 as a DCL specification.

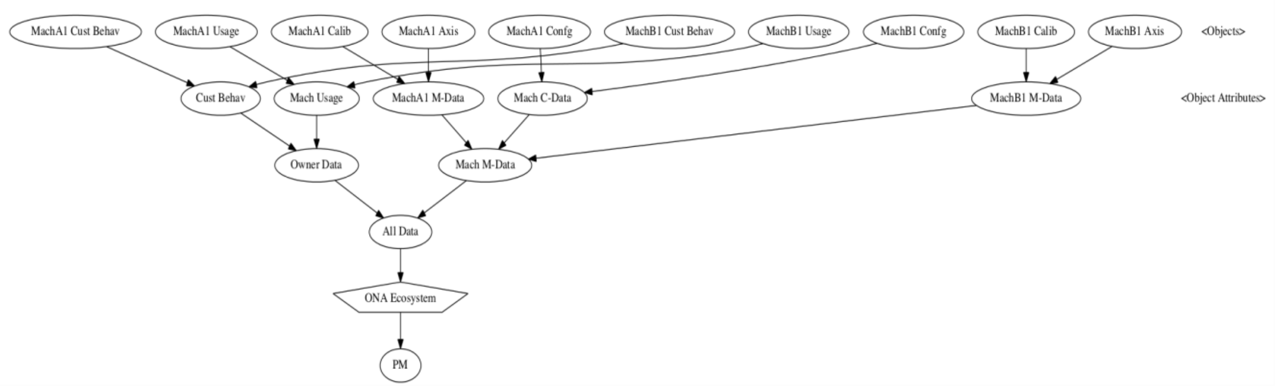


Figure 20: Representative object and object attribute sub-graph of the ONA Ecosystem policy

Table 5: ONA object and object attribute sub-graph policy specification

```
    policy('ONA_Policy_OOA','ONA Ecosystem', [
            object('MachA1 Cust Behav'),
            object('MachA1 Usage'),
            object('MachA1 Calib'),
            object('MachA1 Axis'),
            object('MachA1 Confg'),
            object('MachB1 Cust Behav'),
            object('MachB1 Usage'),
            object('MachB1 Calib'),
            object('MachB1 Axis'),
            object('MachA1 Confg'),

            object_attribute('MachA1 M-Data'),
            object_attribute('MachB1 M-Data'),
            object_attribute('Cust Behav'),
            object_attribute('Mach Usage'),
            object_attribute('Mach M-Data'),
            object_attribute('Mach C-Data'),
            object_attribute('Owner Data'),
            object_attribute('All Data'),

            assign('MachA1 Cust Behav','Cust Behav'),
            assign('MachA1 Usage','Mach Usage'),
            assign('MachA1 Calib','MachA1 M-Data'),
            assign('MachA1 Axis','MachA1 M-Data'),
            assign('MachA1 Confg','Mach C-Data'),
            assign('MachB1 Cust Behav','Cust Behav'),
            assign('MachB1 Usage','Mach Usage'),
            assign('MachB1 Calib','MachB1 M-Data'),
            assign('MachB1 Axis','MachB1 M-Data'),
            assign('MachB1 Confg','Mach C-Data'),
            assign('MachA1 M-Data','Mach M-Data'),
            assign('MachB1 M-Data','Mach M-Data'),
            assign('Cust Behav', 'Owner Data'),
            assign('Mach Usage', 'Owner Data'),
            assign('Owner Data', 'All Data'),
            assign('Mach C-Data', 'Mach M-Data'),
            assign('Mach M-Data', 'All Data'),
            assign('All Data', 'ONA Ecosystem'),

            policy_class('ONA Ecosystem'),
            assign('ONA Ecosystem','PM'),
            connector('PM')
    ]).
```

NGAC “users” (subjects): Definition of users should be added to the ONA Ecosystem policy. These would be the identities of actual individuals who have identities established on the systems being used. Then assignments should be made for each user to appropriate user attributes.

The last steps are to join the user side sub-graph with the object side sub-graph in a single policy file with both sub-graphs ascending from the ONA Ecosystem policy class, and to add the associations between attributes in the two sub-graphs to generate the permissions that conform to the organizational policies.

NGAC “associations”: Table 6 shows the associations for the ONA Ecosystem policy. In this table the portions of the policy specification previously discussed are represented by the six comment lines at the beginning of the policy specification.

Table 6: Associations for the ONA Ecosystem policy specification

```
    policy('ONA_Policy','ONA Ecosystem', [
            % user declarations
            % user attribute declarations
            % assignments to create user sub-graph structure (as above)
            % object declarations
            % object attribute declarations
            % assignments to create object sub-graph structure (as above)

            associate('ONA FEng', [r,w], 'Mach C-Data'), % configuration data
            associate('ONA FEng', [r], 'Mach M-Data'), % mach manufacturing data
            associate('ONA Mgt', [r], 'Owner Data'), % data with restricted access
            associate('CustA Staff', [r], 'MachA1 M-Data'), % cust A owns Mach A1
            associate('CustB Staff', [r], 'MachB1 M-Data'), % cust B owns Mach B1

            policy_class('ONA Ecosystem'),
            assign('ONA Ecosystem','PM'),
            connector('PM')
    ]).
```

The illustrated associations shown:

- provide read and write access to a all machine's configuration data by ONA Field Engineering staff
- provide read access to all machine's manufacturing data by ONA Field Engineering staff
- provide read access to all owner data only by ONA Management staff
- provide read access to the machine manufacturing data of the machine A1 that is owned by customer A to customer A's staff
- provide read access to the machine manufacturing data of the machine B1 that is owned by customer B to customer B's staff

The final assembled policy shown in Table 7 has the graph shown in Figure 21.

Table 7: ONA Ecosystem policy specification

```
    policy('ONA_Policy','ONA Ecosystem', [
            user('Jose'),
            user('Itziar'),
            user('Ian'),
            user('Leandro'),
            user('Rebecca'),

            user_attribute('ONA'),
            user_attribute('ONA Mgt'),
            user_attribute('ONA FEng'),
            user_attribute('ONA Staff'),
            user_attribute('Cust'),
            user_attribute('CustA'),
            user_attribute('CustB'),
            user_attribute('CustA Staff'),
            user_attribute('CustB Staff'),
            user_attribute('3rd Party'),
            user_attribute('3rdP C'),
            user_attribute('3rdP C Staff'),

            assign('Jose','ONA Mgt'),
            assign('Itziar','ONA FEng'),
            assign('Ian','CustA Staff'),
            assign('Rebecca','3rdP C Staff'),
            assign('Leandro','CustB Staff'),

            assign('ONA Mgt','ONA Staff'),
            assign('ONA FEng','ONA Staff'),
            assign('ONA Staff','ONA'),
            assign('3rdP C Staff','3rdP C'),
            assign('3rdP C','3rd Party'),
            assign('CustA Staff','CustA'),
            assign('CustB Staff','CustB'),
            assign('CustA','Cust'),
            assign('CustB','Cust'),
            assign('ONA','ONA Ecosystem'),
            assign('Cust','ONA Ecosystem'),
            assign('3rd Party','ONA Ecosystem'),

            object('MachA1 Cust Behav'),
            object('MachA1 Usage'),
            object('MachA1 Calib'),
            object('MachA1 Axis'),
            object('MachA1 Confg'),

            object('MachB1 Cust Behav'),
            object('MachB1 Usage'),
            object('MachB1 Calib'),
            object('MachB1 Axis'),
            object('MachA1 Confg'),

            object_attribute('MachA1 M-Data'),
            object_attribute('MachB1 M-Data'),

            object_attribute('Cust Behav'),
            object_attribute('Mach Usage'),
            object_attribute('Mach M-Data'),
            object_attribute('Mach C-Data'),

            object_attribute('Owner Data'),
            object_attribute('All Data'),

            assign('MachA1 Cust Behav','Cust Behav'),
            assign('MachA1 Usage','Mach Usage'),
            assign('MachA1 Calib','MachA1 M-Data'),
            assign('MachA1 Axis','MachA1 M-Data'),
            assign('MachA1 Confg','Mach C-Data'),

            assign('MachB1 Cust Behav','Cust Behav'),
            assign('MachB1 Usage','Mach Usage'),
            assign('MachB1 Calib','MachB1 M-Data'),
            assign('MachB1 Axis','MachB1 M-Data'),
            assign('MachB1 Confg','Mach C-Data'),

            assign('MachA1 M-Data','Mach M-Data'),
            assign('MachB1 M-Data','Mach M-Data'),

            assign('Cust Behav', 'Owner Data'),
            assign('Mach Usage', 'Owner Data'),
            assign('Owner Data', 'All Data'),
            assign('Mach C-Data', 'Mach M-Data'),
            assign('Mach M-Data', 'All Data'),
            assign('All Data', 'ONA Ecosystem'),

            associate('ONA FEng', [r,w], 'Mach C-Data'), % configuration data
            associate('ONA FEng', [r], 'Mach M-Data'), % mach manufacturing data
            associate('ONA Mgt', [r], 'Owner Data'), % data with restricted access
            associate('CustA Staff', [r], 'MachA1 M-Data'), % cust A owns Mach A1
            associate('CustB Staff', [r], 'MachB1 M-Data'), % cust B owns Mach B1

            policy_class('ONA Ecosystem'),
            assign('ONA Ecosystem','PM'),
            connector('PM')
    ]).
```

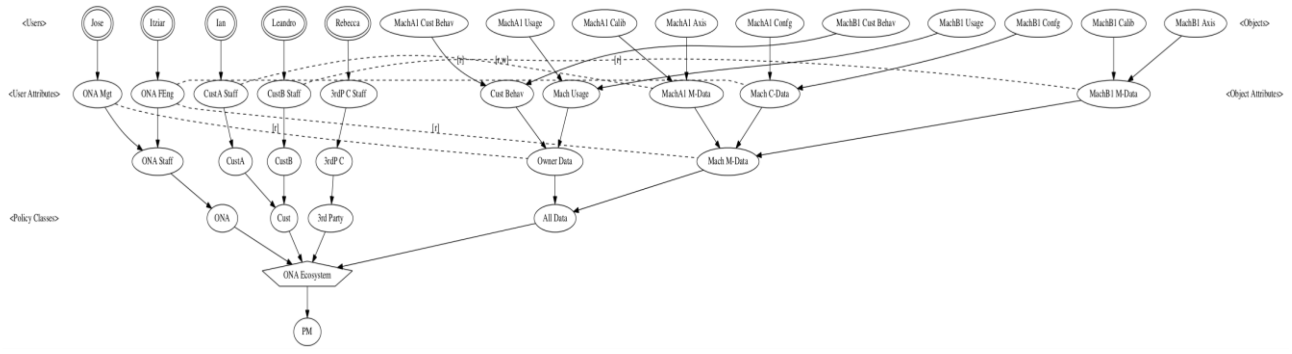


Figure 21: Graph of the ONA Ecosystem policy

15. APPENDIX D –NGAC APPLIED TO IISF SECURITY FUNCTIONS

The IISF covers virtually all aspects of security, both technical (that is, mechanisms that are implemented as hardware and/or software) and non-technical (that is, policies, procedures, and practices). Figure 22 shows the *functional viewpoint* of the IISF as six interacting building blocks, organized as three layers. The top layer represents the four core security functions, which are in turn supported by a data protection layer and a security model and policy layer.

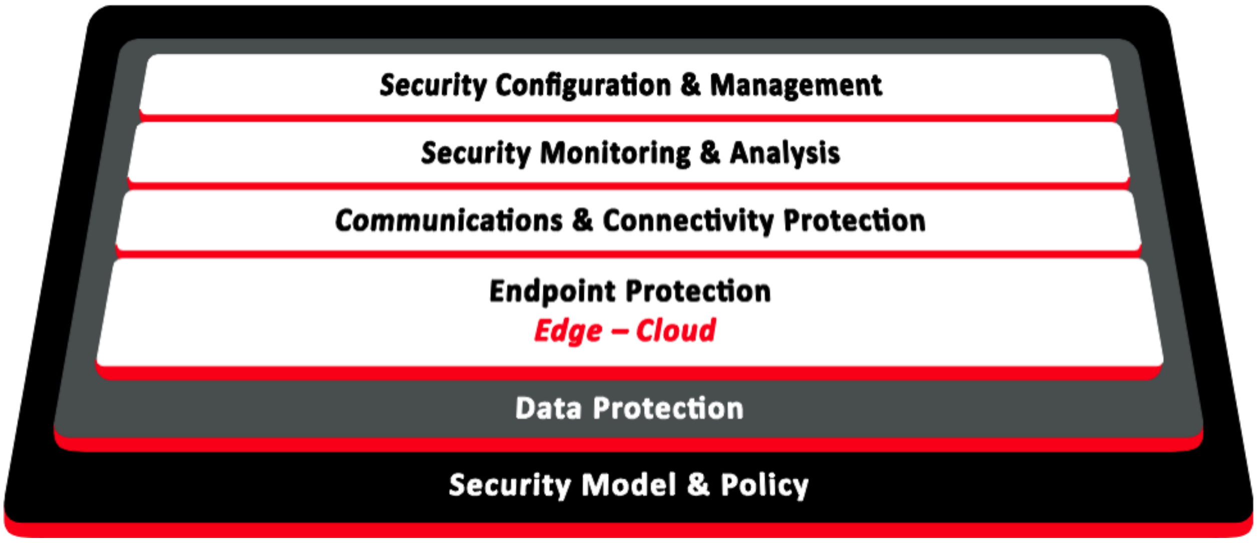

Security Configuration & Management
Security Monitoring & Analysis
Communications & Connectivity Protection
Endpoint Protection
Edge – Cloud
Data Protection
Security Model & Policy

Figure 22: Functional Building Blocks of the IISF

15.1. OVERVIEW OF NGAC APPLICABILITY IN THE IISF

NGAC is concerned with security model and policy, so in Figure 23 we indicate, is a very coarse way, the role of NGAC relative to the functional building blocks of the IISF. In this and subsequent figures a solid green ellipse indicates a primary role, while a dashed ellipse indicates a secondary or optional role.

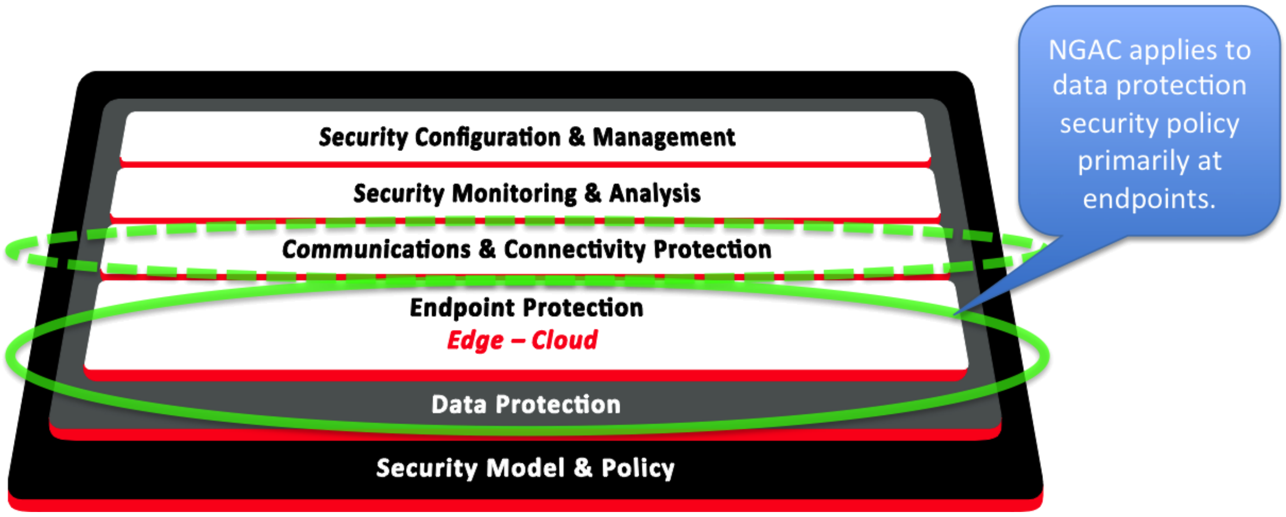

NGAC applies to data protection security policy primarily at endpoints.
Security Configuration & Management
Security Monitoring & Analysis
Communications & Connectivity Protection
Endpoint Protection
Edge – Cloud
Data Protection
Security Model & Policy

Figure 23: Overview of NGAC role in IISF functional building blocks

15.2. Security Model and Policy

Figure 24 illustrates role of NGAC within the functional breakdown for the security model and policy within IISF. The left side of this figure, Security Policy, is primarily non-technical and refers to organizational security goals and objectives. The right side, Security Model, is the technical aspect. NGAC is a technical measure providing security policy (mode) specification and enforcement. Thus, we show the emphasis of NGAC on the technical side. NGAC provides the capability to express and enforce a security model primarily at the end points. However, since NGAC is distributed, it depends on communications security. However, it is not primarily used to express or enforce communications and connectivity security policy as described by the NGAC standard. The following analysis will present a more refined explanation of the role of NGAC in the IISF.

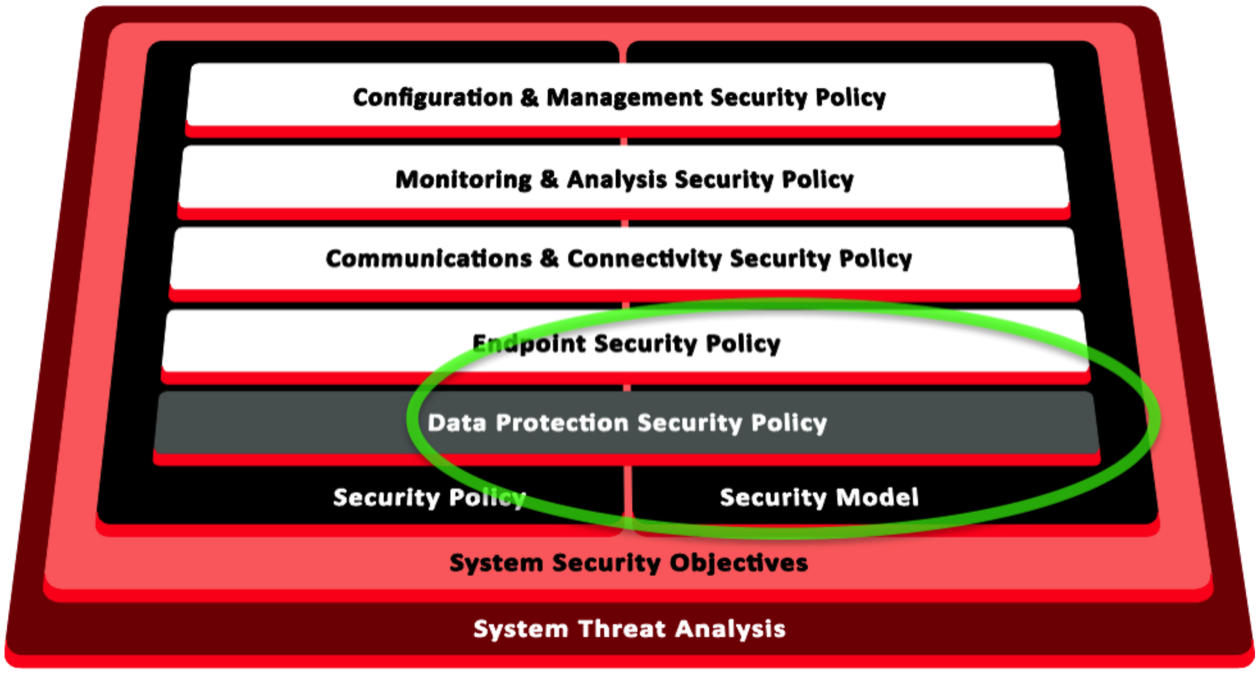

Configuration & Management Security Policy
Monitoring & Analysis Security Policy
Communications & Connectivity Security Policy
Endpoint Security Policy
Data Protection Security Policy
Security Policy
Security Model
System Security Objectives
System Threat Analysis

Figure 24: Overview of NGAC role in IISF security model and policy functional breakdown

15.3. Endpoint Protection

Figure 25 shows the IISF functional breakdown for endpoint protection and the role that NGAC can play in endpoint data protection. NGAC is not currently supported natively within ubiquitous operating systems, e.g. through a “pluggable authorization module.” The current feasible deployments of NGAC depend on the endpoint’s operating system to provide the basic security properties of isolation and integrity for the resources that it exports, and which are placed under the NGAC scope of control. Thus, the role of endpoint security model and policy is shared with the operating environment (OE); for this reason a dashed ellipse is used to indicate NGAC’s role in this aspect. NGAC can provide fine-grained access control to OE-exported resources, and can thus provide fine-grained integrity protection, in the sense of “no unauthorized modification”, for objects under its scope of control.

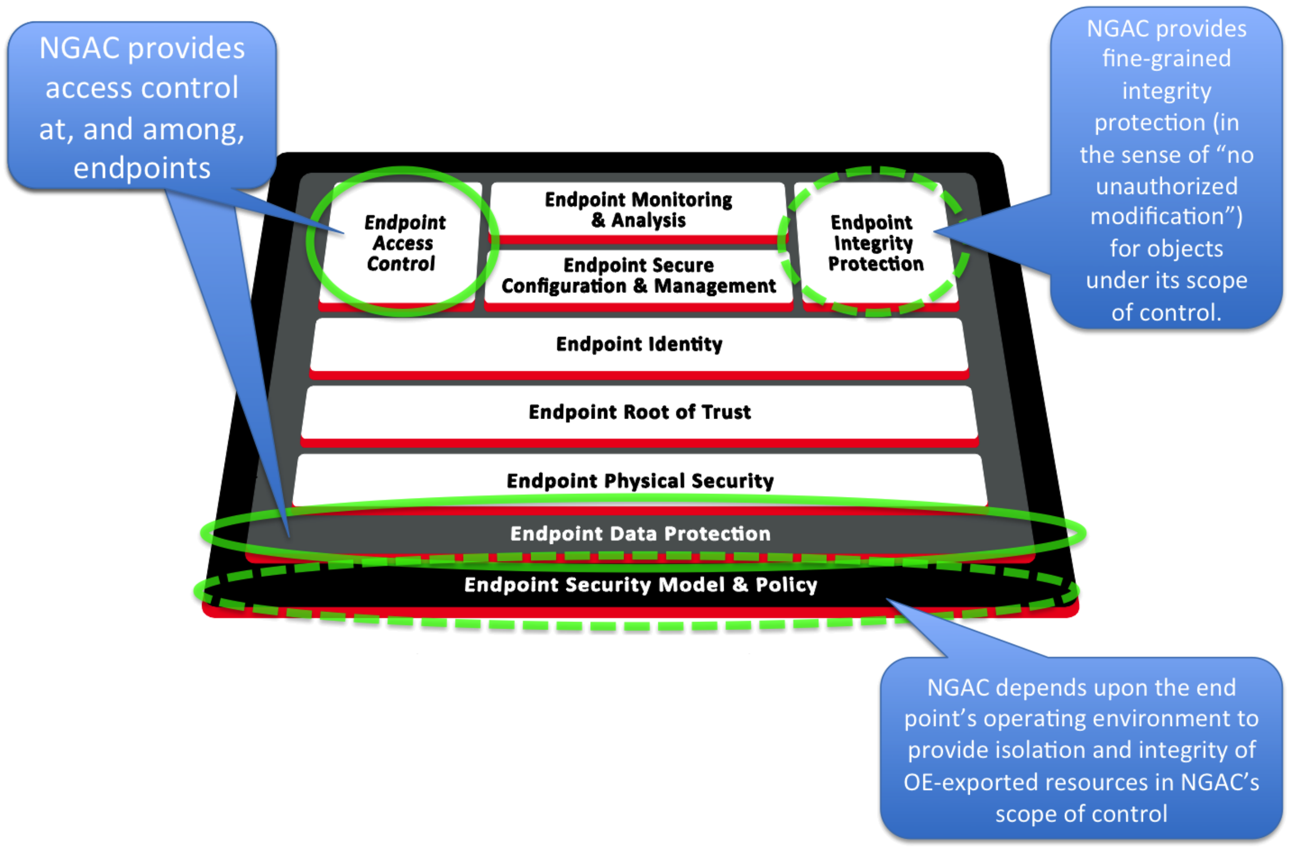

NGAC provides access control at, and among, endpoints
Endpoint Access Control
Endpoint Monitoring & Analysis
Endpoint Secure Configuration & Management
Endpoint Integrity Protection
NGAC provides fine-grained integrity protection (in the sense of "no unauthorized modification") for objects under its scope of control.
Endpoint Identity
Endpoint Root of Trust
Endpoint Physical Security
Endpoint Data Protection
Endpoint Security Model & Policy
NGAC depends upon the end point's operating environment to provide isolation and integrity of OE-exported resources in NGAC's scope of control

Figure 25: NGAC role in endpoint protection

15.4. Communication and Connectivity Protection

Concerning communication and connectivity protection, the functional breakdown of which is illustrated in Figure 26, NGAC does not have a primary role as it is defined in the NGAC standard. It could be used to model permissions for endpoints to communicate, and we are investigating the use of NGAC's policy modeling capabilities to represent information flow control, but other aspects of communication security are outside the scope of NGAC.

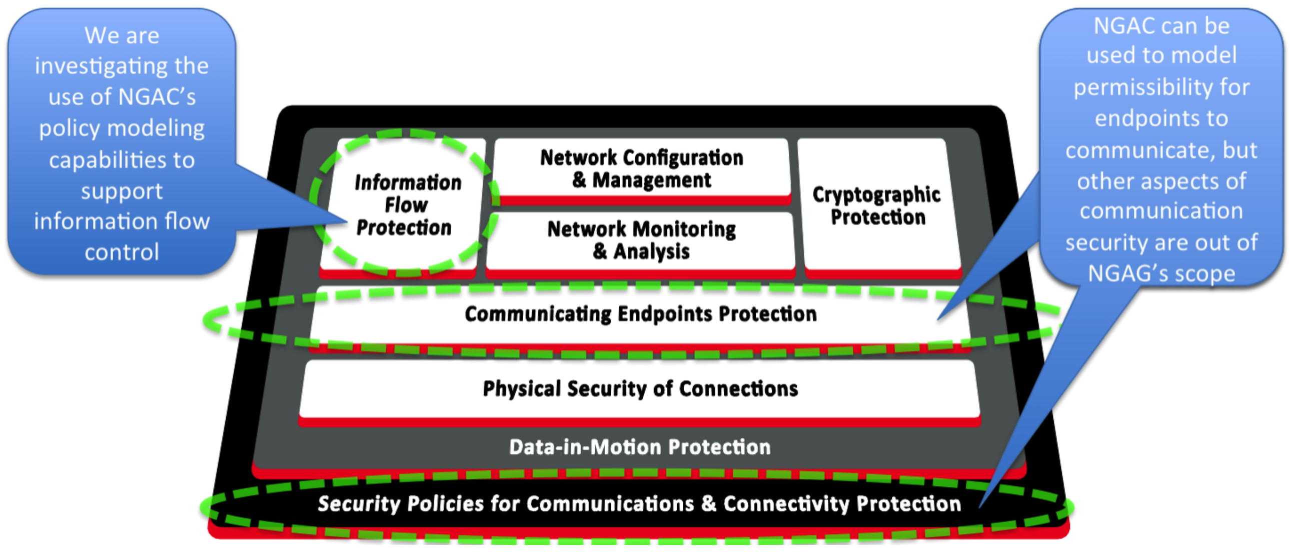

We are investigating the use of NGAC's policy modeling capabilities to support information flow control
Information Flow Protection
Network Configuration & Management
Network Monitoring & Analysis
Cryptographic Protection
NGAC can be used to model permissibility for endpoints to communicate, but other aspects of communication security are out of NGAG's scope
Communicating Endpoints Protection
Physical Security of Connections
Data-in-Motion Protection
Security Policies for Communications & Connectivity Protection

Figure 26: NGAC role in communication and connectivity protection

15.5. SECURITY MONITORING AND ANALYSIS

Figure 27 illustrated the IISF functional breakdown of security monitoring and analysis. NGAC does not cover this aspect of security. It is conceivable that NGAC could play a supporting role for this aspect, depending on the level of abstraction at which monitoring is conceptualized and implemented, by protecting monitoring resources and granting use of those resources to monitoring agents. In this regard, monitoring is like NGAC itself because of its dependence on the operating environment to provide the underlying allocation of protection of resources used for monitoring. Whether monitoring resources are placed within the NGAC scope of control could be influenced by the consideration whether monitoring and its resources should be part of the overall access control policy, or weather it should be treated as a distinct subsystem from NGAC.

NGAC does not cover the Security Monitoring and Analysis function.

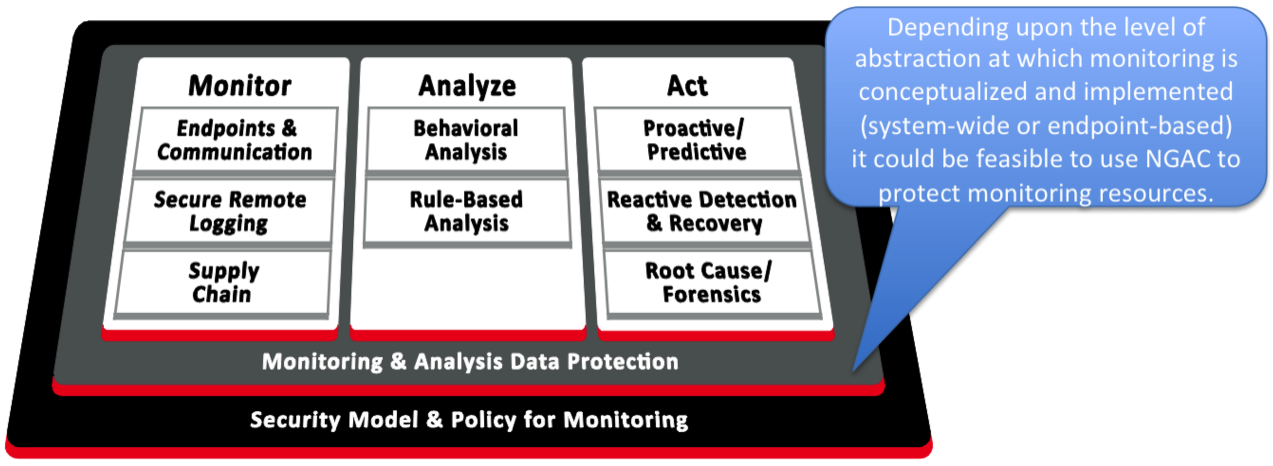

Monitor
Endpoints & Communication
Secure Remote Logging
Supply Chain
Analyze
Behavioral Analysis
Rule-Based Analysis
Act
Proactive/ Predictive
Reactive Detection & Recovery
Root Cause/ Forensics
Monitoring & Analysis Data Protection
Security Model & Policy for Monitoring
Depending upon the level of abstraction at which monitoring is conceptualized and implemented (system-wide or endpoint-based) it could be feasible to use NGAC to protect monitoring resources.

Figure 27: NGAC role in monitoring and analysis

15.6. SECURITY CONFIGURATION AND MANAGEMENT

Figure 28 shows the application of NGAC in the context of the IISF's functional breakdown for security configuration and management. The primary application of NGAC is its natural role in controlling changes to security policy models as is recommended for NGAC-based enforcement mechanisms. It can also potentially be used for configuration and management data protection if effectively integrated with the operating environment and other management functions. NGAC could also be used for change control of security configuration data used by other security enforcement mechanisms, thus further unifying security management.

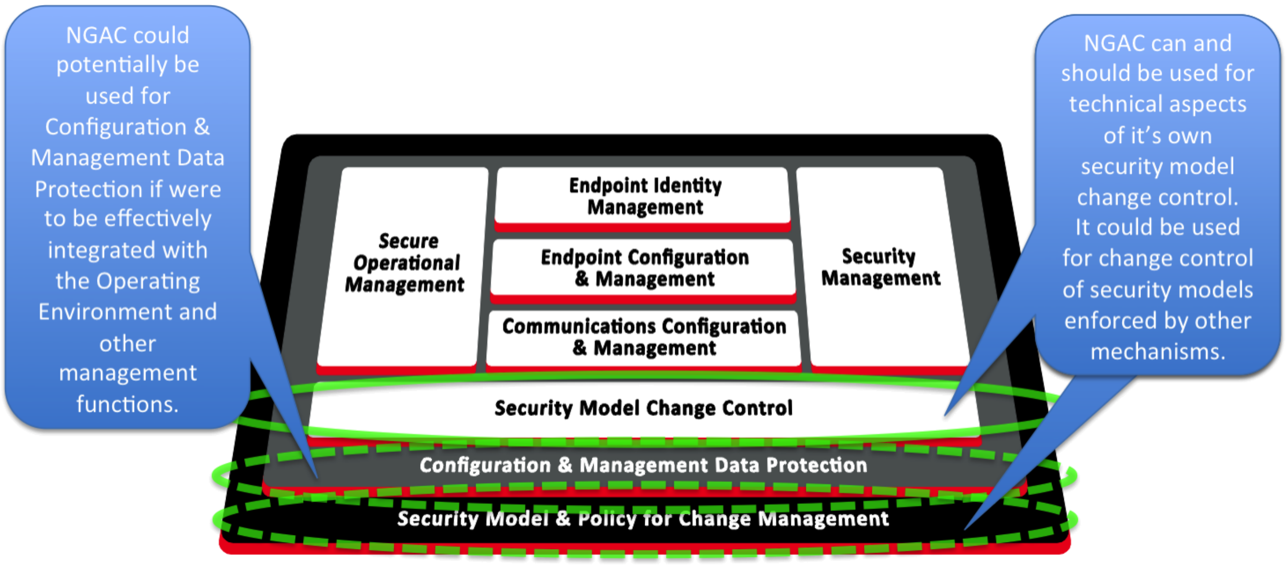

NGAC could potentially be used for Configuration & Management Data Protection if were to be effectively integrated with the Operating Environment and other management functions.
NGAC can and should be used for technical aspects of it's own security model change control. It could be used for change control of security models enforced by other mechanisms.
Secure Operational Management
Endpoint Identity Management
Endpoint Configuration & Management
Communications Configuration & Management
Security Management
Security Model Change Control
Configuration & Management Data Protection
Security Model & Policy for Change Management

Figure 28: NGAC role in security configuration and management

15.7. DATA PROTECTION

Figure 29 shows the role of NGAC in the functional breakdown of data protection. The primary application of NGAC is to provide security model and policy for endpoint data protection; this is NGAC's *raison d'être*. Because NGAC is not yet integrated into the operating environment, current NGAC implementations leverage the OE to build mechanisms that enforce NGAC policies. As mentioned with respect to security configuration and management, NGAC can be used to protect all manner of operational and security-related data, including communications-related data, configuration data, and monitoring data. With appropriate policies and extensions to enforcement mechanisms, NGAC can address a *slice* of data protection across the system, including data-at-rest (DAR), data-in-use (DIU), and data-in-motion (DIM), providing a higher-level abstraction of protections provided by the operating environment and the network hardware and software.

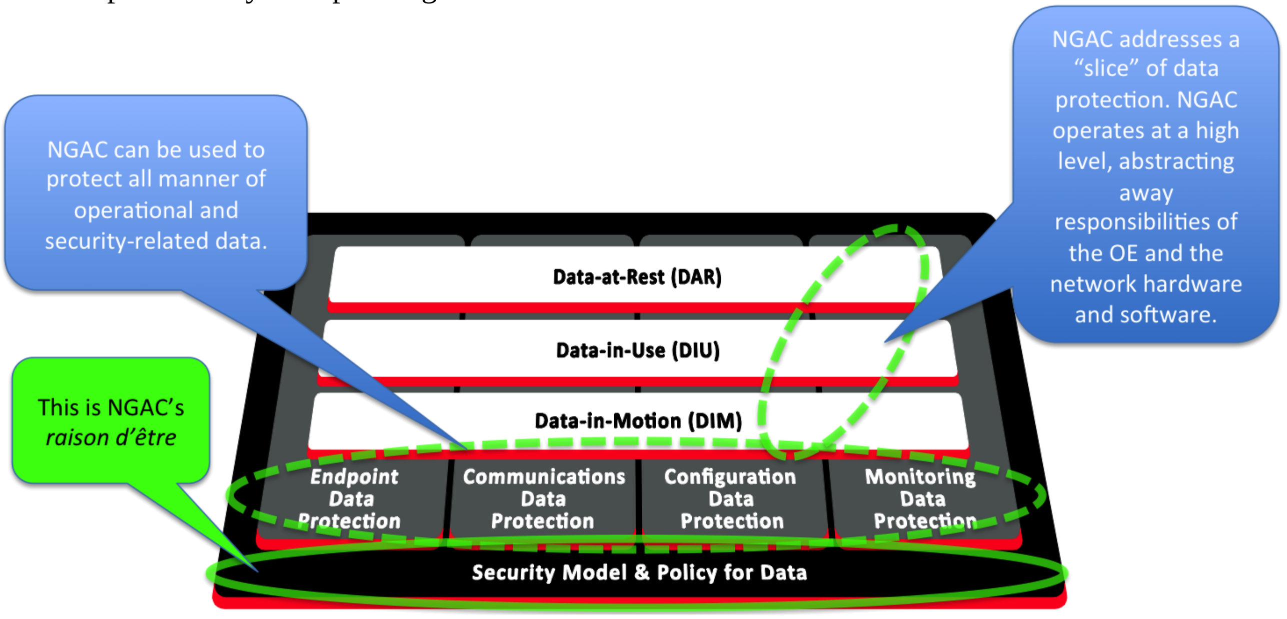

NGAC addresses a "slice" of data protection. NGAC operates at a high level, abstracting away responsibilities of the OE and the network hardware and software.
NGAC can be used to protect all manner of operational and security-related data.
This is NGAC's *raison d'être*
Data-at-Rest (DAR)
Data-in-Use (DIU)
Data-in-Motion (DIM)
Endpoint Data Protection
Communications Data Protection
Configuration Data Protection
Monitoring Data Protection
Security Model & Policy for Data

Figure 29: NGAC role in data protection

15.8. REFINED ROLE OF NGAC IN THE IISF

Having examined the role, and potential roles, of Next Generation Access Control in various aspects of security within the Industrial Internet Security Framework, we now present in Figure 30 a refined view of NGAC's roles in the functional breakdown for security model and policy. Previously, in Figure 24 we identified in a broad sense the applicability of NGAC as centering on data protection security policy, endpoint security policy, and security model.

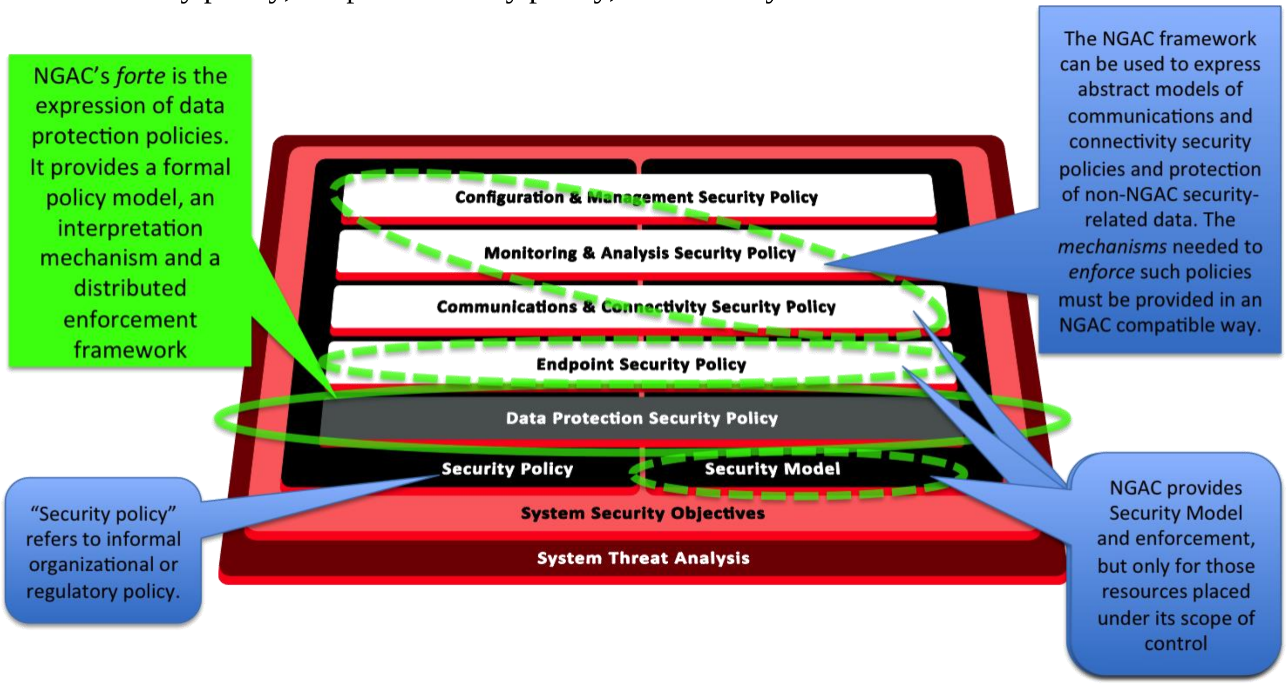

NGAC's *forte* is the expression of data protection policies. It provides a formal policy model, an interpretation mechanism and a distributed enforcement framework
The NGAC framework can be used to express abstract models of communications and connectivity security policies and protection of non-NGAC security-related data. The *mechanisms* needed to *enforce* such policies must be provided in an NGAC compatible way.
Configuration & Management Security Policy
Monitoring & Analysis Security Policy
Communications & Connectivity Security Policy
Endpoint Security Policy
Data Protection Security Policy
Security Policy
Security Model
System Security Objectives
System Threat Analysis
"Security policy" refers to informal organizational or regulatory policy.
NGAC provides Security Model and enforcement, but only for those resources placed under its scope of control

Figure 30: NGAC role in security model and policy refined

Here we assert NGAC's *forte* as the expression of data protection policies, providing a formal framework for policy models, an interpretation mechanism and a distributed enforcement framework in its reference implementations. We call attention to the fact that the language of the IISF is somewhat nuanced in its distinction between security policy and security model, using "security policy" to refer to informal organizational or regulatory policy, and "security model" to refer to the more precise and machine process-able representations that NGAC refers to a "policy specification". In this sense, NGAC provides security model and enforcement, but only for those resources that are placed under its scope of control through an appropriate configuration of the underlying protection mechanisms of the operating environment.

From the standpoint of policy specification the NGAC framework can be used to express abstract *policy models* of communications and connectivity security policies and protection of non-NGAC security-related data. As we have described in the foregoing presentation, such applications may include configuration & management security policy, monitoring & analysis security policy, communications & connectivity security policy, in addition to endpoint security policy. The *mechanisms* needed to enforce such additional policies must be provided in and NGAC-compatible way by creating policy enforcement points (PEPs) appropriate to the new resources and appropriate resource access points (RAPs) for those resources. These PEPs may call upon the NGAC policy decision point(s) (PDPs) to render access control verdicts based on the policies stored in the policy information point (PIP).