European Middleware Initiative

Argus EES - Execution Environment Service

Document version:	**1.0.0**
EMI Component Version:	**0.0.10**
Date:	**April 28, 2011**

This work is co-funded by the EC EMI project under the FP7 Collaborative Projects Grant Agreement Nr. INFSO-RI-261611.

Execution Environment Service (EES)

FUNCTIONAL DESCRIPTION The EES is a pluggable, configurable authorisation service similar to the Site Central Authorisation Service (SCAS). The role of the EES is to ensure that an appropriate site-specific execution environment is procured based on the site-agnostic obligations and attributes it receives as input in the form of SAML2-XACML2 requests. It runs as a standalone service, responding to requests from a Policy Enforcement Point (PEP) which have been augmented with information from a Policy Decision Point (PDP).

From the outside, the EES can be viewed as an obligation transformer; for example it can be used to transform a site-agnostic obligation for a local account mapping to a site-specific obligation for on-demand virtual machine deployment.

To integrate the EES with an existing Argus installation, a separate component called the EES Obligation Handler should be configured in the PEP daemon. For more details regarding integration in Argus, please see the documentation for this component. The EES itself ships with a pre-configured transformer plug-in which extracts PDP data from the SAML2-XACML2 environment attributes. This plug-in is not required when PDP data is not transmitted to the EES.

DAEMONS RUNNING ${prefix}/sbin/ees

INIT SCRIPTS AND OPTIONS (start|stop|restart|reload|status) ${prefix}/etc/ees

CONFIGURATION FILES WITH EXAMPLE OR TEMPLATE The EES is designed to be highly customizable. Its configuration model allows policies to be expressed as state machines in the Policy Description Language (PDL), whose branches end in pre-configured plug-in instances. A small example as well as an ees.conf manpage is provided.

${prefix}/etc/ees.conf

LOGFILE LOCATIONS (AND MANAGEMENT) AND OTHER USEFUL AUDIT INFORMATION Syslog available: yes No custom log file configurable yet

OPEN PORTS 6217

POSSIBLE UNIT TEST OF THE SERVICE A high-level test script (test_ees.sh) is available.

WHERE IS SERVICE STATE HELD The EES uses plug-ins to connect to various other middleware. The configuration file for the EES defines the plug-ins used, as well as any dependant configuration files such as /etc/grid-security/gridmapfile and gridmapdir.

An integral part of the EES is the Attribute and Obligations Store (AOS), which is a component that allows plug-ins to query the (transient) SAML2-XACML2 data received. This object store is exposed through a simple API. This data is logged, but the intermediate state is not saved.

CRON JOBS None.

SECURITY INFORMATION The EES should run firewalled from the rest of the network, only allowing the Argus PEPd access.

ACCESS CONTROL MECHANISM DESCRIPTION (AUTHENTICATION & AUTHORIZATION) Mandated by plug-in and network configuration.

HOW TO BLOCK / BAN A USER Through Argus.

NETWORK USAGE Exposes a SOAP service that transforms SAML2-XACML2 requests.

FIREWALL CONFIGURATION The EES currently has no support for TLS connections. System administrators should configure the EES host to only allow access to the EES from the PEPd host.

SECURITY RECOMMENDATIONS

SECURITY INCOMPATIBILITIES

LIST OF EXTERNAL PACKAGES SAML2-XACML2-C-LIB

```
OTHER SECURITY RELEVANT COMMENTS

UTILITY SCRIPTS

LOCATION OF REFERENCE INFORMATION FOR USERS Argus documentation

LOCATION OF REFERENCE INFORMATION FOR ADMINISTRATORS
https://www.nikhef.nl/pub/projects/grid/gridwiki/index.php/EES
```

NAME

ees − An XACML webservice to create execution environments

SYNOPSIS

ees [*configfile*]

DESCRIPTION

ees runs the EES in the background unless debug mode was selected at compile time.

COMMAND−LINE OPTIONS

configfile

Will run the EES with the specified config.

SEE ALSO

ees.conf(5)

AUTHORS

ees was written by Aram Verstegen <aramv@nikhef.nl>, with help from Oscar Koeroo <okoeroo@nikhef.nl> and Mischa Salle <msalle@nikhef.nl>.

NAME

ees.conf – **ees(1)** configuration file

OVERVIEW

The ees configuration file language allows you to create logical trees of execution for EES configured plug-ins.

EXAMPLES

```
plugin_1 = "ees_plugin1.mod"
        "argv[1]"
        "argv[2]"

plugin_2 = "ees_plugin2.mod"

plugin_3 = "ees_plugin3.mod"

# Will execute plugin 1, 2 and 3 if each previous plugin run was successful policy_1:
  plugin_1 -> plugin_2
  plugin_2 -> plugin_3

# Only runs plugin 2 if plugin 1 was successful policy_2:
  plugin_1 -> plugin_2
  plugin_3

# Try plugin 2 after plugin 1, if plugin 2 fails, run plugin 3 and try plugin 2 again policy_3:
  plugin_1 -> plugin_2 | plugin_3
  plugin_3 -> plugin_2
```

SEE ALSO

ees(1)

Table of contents

1. INTRODUCTION

1.1. PURPOSE

The purpose of this document is to give a detailed description of the EES, the Execution Environment Service, with a focus on its external interfaces.

1.2. DOCUMENT ORGANISATION

This document is organized as follows

- Section 2 gives a short description of the requirements of the EES.
- Section 3 explains the context and responsibilities of the EES, and provides an overview of its various components.
- Section 4 documents the APIs of the EEF, AOS and plug-ins insofar they are exposed to external developers.
- Section 5 outlines an example plug-in module.

1.3. APPLICATION AREA

The audience for this document is developers of other AuthZ-FW components, such as the PDP, and of the plug-in modules doing the actual work. The description of the internal implementation of the EES is therefore beyond the scope of this document.

1.4. REFERENCES

Table 1: Table of references

R 1	A. Anderson et al. *SAML 2.0 profile of XACML v2.0*, OASIS Standard, 1 February 2005, *http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-saml-profile-spec-os.pdf*
R 2	C. La Joie, *SOAP Profile for XACML-SAML*, Working Draft 01, 30 November 2007, *http://switch.ch/grid/support/documents/xacmlsaml.pdf*
R 3	*Authorization Service Design*, 29 August 2008, *https://edms.cern.ch/file/944192/1/EGEE-III-JRA1-Security-Design_v1.1.pdf*
R 4	*EGEE Document Management Procedure, http://project-egee-iii-na1-qa.web.cern.ch/project-EGEE-III-NA1-QA/EGEE-III/Procedures/DocManagmtProcedure/DocMngmt.htm*
R 5	*EGEE glossary, http://glossary.eu-egee.com/index.php?id=368*
R 6	*POSIX / SUSv3, http://www.unix.org/version3/online.html*
R 7	*Policy Description Language (PDL), http://www.nikhef.nl/grid/lcaslcmaps/pdl_requirements.pdf*
R 8	*RFC2307: An Approach for Using LDAP as a Network Information Service http://datatracker.ietf.org/doc/rfc2307/*
R 9	*RFC2616: Hypertext Transfer Protocol – HTTP/1.1 http://datatracker.ietf.org/doc/rfc2616/*
R 10	*RFC2818: HTTP Over TLS http://datatracker.ietf.org/doc/rfc2818/*

1.5. DOCUMENT AMENDMENT PROCEDURE

Amendments, comments and suggestions should be sent to the EGEE Authorization Service Framework team mailinglist using the email address *project-egee-authz-service@cern.ch*.

The procedures are documented in the EGEE “Document Management Procedure” [R 4].

1.6. TERMINOLOGY

This subsection provides the definitions of terms, acronyms, and abbreviations required to properly interpret this document. A complete project glossary is provided in the EGEE glossary [R 5].

Glossary

AOS	Attribute and Obligation Store
Argus	The new gLite Authorization Framework, based on the Oasis XACML standard
EEF	EES Execution Framework
EES	Execution Environment Service
EI	EES Interface
EIC	EES Interface Component
LCMAPS	Local Credential MAPping Service
OASIS	Organization for the Advancement of Structured Information Standards
Obligation Handler	Function that handles an obligation triggered by a specific ObligationId in an XACML response message
PAP	Policy Administration Point
PDL	Policy Description Language
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PEPd	Policy Enforcement Point Daemon
PIP	Policy Information Point
SAML	Security Assertion Markup Language
XACML	eXtensible Access Control Markup Language

2. REQUIREMENTS OF THE EES

2.1. PURPOSE

The purpose of the EES is to provide an appropriate site-specific execution environment based on site-agnostic obligations and attributes it receives as input in the form of SAML2-XACML2 requests that have been processed by the PDP for authorization. This service is developed as part of Argus, the new gLite authorization framework, and as such will be communicating primarily with the PEP daemon, but can also act as a standalone service.

2.2. SCOPE

The current EES design is focused on Unix- and POSIX-like execution environments. It does not address non-POSIX environments, such as Microsoft Windows, although the basic design is portable to any system.

2.3. REQUIREMENTS

Because the EES is a standalone service with various interaction interfaces, it can be developed independently from other components in the Argus framework. As such, the language in which the service is written is an implementation-specific internal detail. It should be able to satisfy the following criteria, in no particular order:

- Should be portable across gLite supported platforms, i.e. POSIX
- Should be easy to maintain
- Should be able to serve concurrent requests
- Should perform its operations fast
- Should have a low memory footprint
- Should be able to use required (third party) dependencies
- Should be able to run with reduced privileges
- Functionality should be extendable through plugins
- Should allow interaction with, for example:
    - Local Resource Management Systems (LRMS), either via direct API invocation or through scripting
    - Virtual Machine (framework)

2.4. PRECONDITIONS

Before the EES is called, we assume that:

- All attributes sent to the EES are verified and authentic.

  This should be handled by either the PDP or by the PEPd. If this assumption does not hold, it would be possible to do additional processing by adding a verification plug-in module to the EES. In that case, attributes must be processed by this plug-in, which must verify and authenticate all incoming attributes before other plug-ins run. Adding such a module is not preferred when the EES is part of a full AuthZ-FW.
- Both (user) credentials and the task description have passed the site’s effective policy.

  No additional policy information must be contained in the EES, as this would either duplicate information with the risk of inconsistency, or give a site multiple control points. The EES may of course be incapable of procuring the correct environment, at which point a *NotApplicable* decision shall be returned.

  The current design of the AuthZ-FW already ensures this precondition is met.

Furthermore, we recognize the following additional requirements:

- An optional list of 'supported obligations' by the enforcing PEP can be passed to the EES in the environment of the message. When present, it is used by the EES and its modules to make sure an environment compatible with the target task is created. It is also used to - in effect - 'negotiate' the list of supported obligations, so that 'more advanced' PEP clients can be served by older-version EES's and vice versa.

  If no such list is provided the EES will function, but may return obligations in a format that cannot be processed by the PEP, even though obligations representing the same semantic meaning in a different format could have been returned otherwise.
- Some modules will require that specific logical share selection has been done beforehand by the PDP, based on the information available elsewhere in the AuthZ FW, i.e. in the PAP. The effective policy on the logical share should be decided before the EES is invoked. If this is not the case, the EES will need to use specific attributes to decide. The latter behaviour is ultimately not a task for the EES.

2.5. POSTCONDITIONS

The EES returns a SAML2-XACML2 (see Section 2.6) response message that includes

- A **decision**. The decision will be *Allow* if an execution environment was procured without incident. The decision will be *NotApplicable* in case the requested environment could not be procured. The decision will be *Deny* in case the user is explicitly banned by the site's effective policy. When operating as part of Argus, the EES should never reach a decision of *Deny.*
- A **list of obligations**. It is up to the receiving PEP to either enforce these obligations in an appropriate way or to effectively deny access (since it cannot enforce a required obligation).

2.6. COMMUNICATION PROTOCOLS

The EES is a service processing SAML2-XACML2 messages that comply with the SAML2-XACML2 profile as defined in the *SAML 2.0 profile of XACML [R 1]*.

The plug-ins within the EES should be able to receive obligations that are communicated to the PEP by the PDP. The PEP will send a request message to the EES and the EES communicates back a response message. Unfortunately, since the XACML standard does not support expressing Obligations in a request message, we propose to recreate the received Obligations in a request message by using specific attribute fields in the Environment context. To discern these attributes from the real Environment attributes the original Obligation attributes can be prefixed with a unique identifier when added to the Environment context.

The EES itself is agnostic about communicated attributes. It will process valid SAML2-XACML2 messages and make the attributes they contain available to implementation-specific plug-ins.

Clients *external* to the EES service host will communicate with the EES using mutually authenticated secure channels.

2.7. PRIVILEGE SEPARATION

The service will support privilege separation, e.g. by running the network communication and the processing of the configuration files in a non-privileged context, while executing only a selected set of plugins with elevated privileges. Elevated privileges can range from a dedicated or specially assigned group and/or account to administrative (i.e. root) privileges which will be dropped during normal operations.

There are use cases that require a specific system group or account to be enforced before the EES is able to perform the requested task. For example Maui requires a specific group to be associated with the process before allowing access to the Maui API. However it might be undesirable to run the entire service as a member of this group. Other use cases are the possibility to hide credentials for e.g. an LDAP password file or certificate for client authentication to LDAP or other service, such as OpenNebula.

Switching of privileges can be achieved using the *seteuid()* and *setegid()* system calls specified by POSIX. These system calls provide the option to switch effective and real privilege contexts, which means privileges that were initially dropped can be restored when needed at a later time. To make use of the *seteuid()* and *setegid()* calls for privilege separation, the EES needs to be started with effective administrative (i.e. root) privileges.

User switching system calls are not well-defined in a multi-threaded environment. Hence, in the concurrency model used by the EES, this functionality is only available when plugins are executed exclusively (i.e. in the `plugin_initialize()` function). This could be mitigated by making use of a `sudo`- or `suexec`-like wrapper binary. See also Section 3.6.2.

3. DESCRIPTION OF THE EES

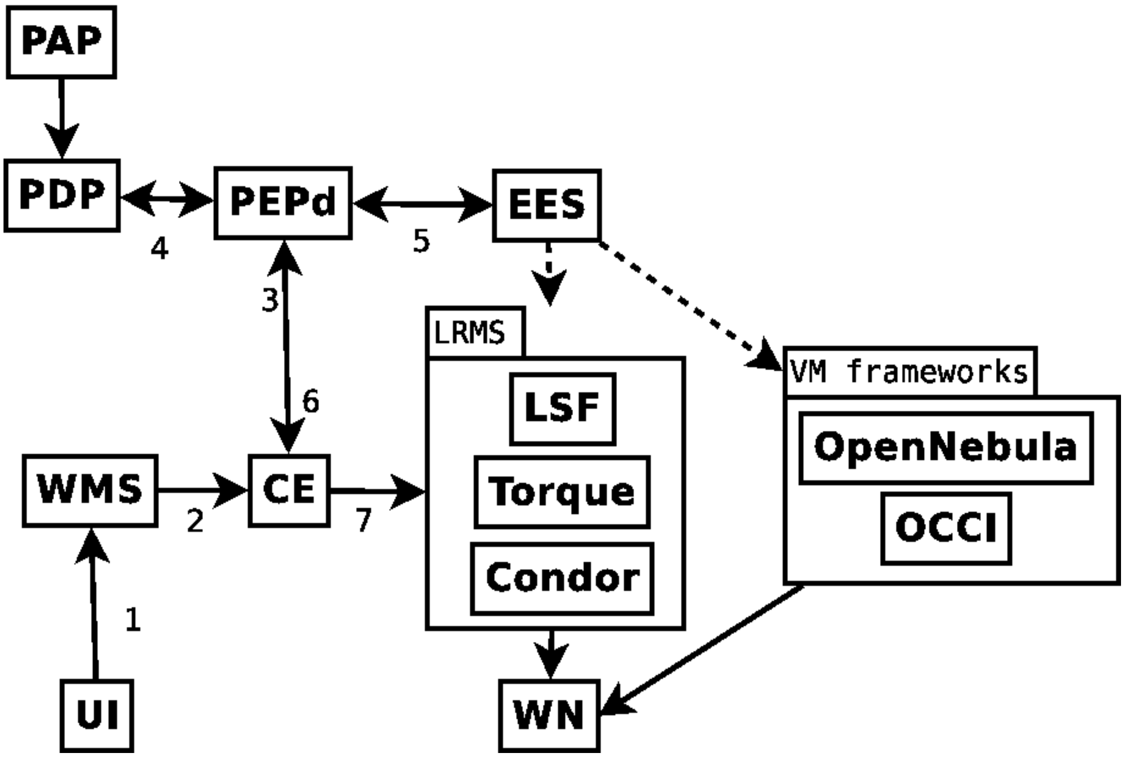

Figure 1: Role of the EES in Argus, in the context of VM scheduling

The role of the EES is to ensure that an appropriate site-specific execution environment can be procured that allows an already-authorized task to be executed on a site-local resource. The EES can be used to provision these execution environments in the role of an Obligation Handler, or can act as a Policy Information Point (PIP). In Figure 881 we outline the role of the EES within a typical Argus setup.

From the ‘outside’ the EES appears as an ‘obligation transformer’. It takes execution-agnostic attributes and obligations and makes sure that

- an environment which honours the effective policy is procured;
- new obligations are created that ensure the PEP can properly move the task into the procured execution environment;
- any agnostic obligations that have been fully translated into a site-specific obligation are removed, since the PEP has to be able to enforce all obligations present.

Examples of this procurement may be:

- preparing or launching a virtual machine on a worker node;
- the assignment of a site-local Unix *UID* or user name out of a pool based on the list of FQANs (Fully Qualified Attribute Name) given by the VO;
- the assignment of a site-local primary Unix *GID* or group name based on the obligation to run a job in a particular logical *share;*
- the assignment of a site-local LDAP group membership.

3.1. COMPONENTS OF THE EES

The EES is composed of the following components, see also Figure 2 and Sections 3.2 through 3.5:

- The **EES Interface (EI)** – comprised of a set of **EES interface components (EICs)**, which can access the functionality contained within the EEF. For now these components include binding to Unix domain- or HTTP(S) sockets. The EICs all use the EEF core API to invoke the EES functionality, triggering the creation of a separate thread for each call.
- The **EES Execution Framework (EEF) –** A thread-safe core 'execution framework' executing the business logic of the EES. This consists of:
    - The **Attribute and Obligation Store (AOS)** – a common store of attributes and obligations that is used by the EEF to maintain internal state between the handling of a request message and formulating the response.
    - The **Evaluation Manager (EM) –** A component that drives the business logic within the EEF based on a configuration of plug-in modules. The set of rules resulting from the configuration constitutes a site-local mapping *policy.*
    - The **Plugin Manager (PM) –** This component will load, run and eventually unload a set of plug-in modules based on the effective policy in the Evaluation Manager (EM).
    - A set of thread-safe '**plug-in modules**', each of which procures part of an execution environment and in the process update the state of the thread-local AOS when needed.

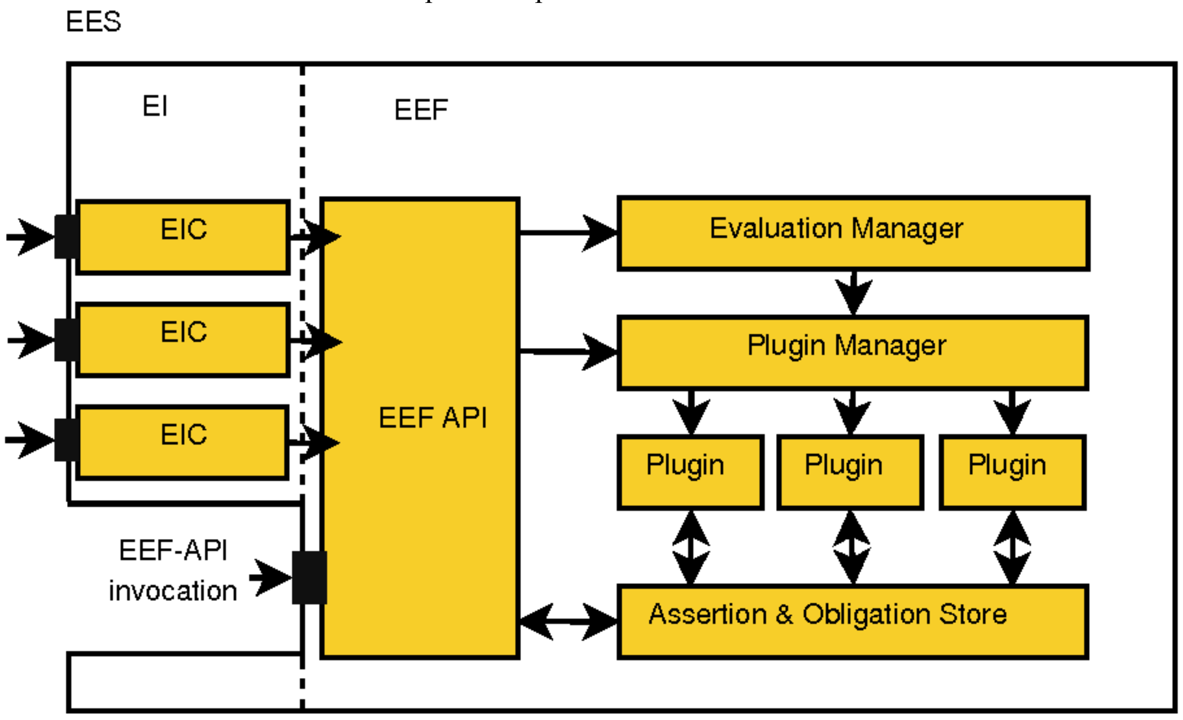

Figure 2: outline of the EES components

3.2. EES INTERFACE COMPONENTS: EICS

The EES can be invoked in several different ways, all resulting in the execution of the EES Execution Framework (EEF). The different ways are implemented using loosely coupled drivers which will be described in detail in Section 4.2. Here we just mention their basic functionality.

- Unencrypted networked invocation

  We foresee that this invocation is only used in a trusted environment, for example when Argus is deployed on a single host while Unix domain sockets or named pipes would not be an option.

- Networked invocation over a TLS/SSL secure channel

  This is similar to the previous invocation but using TLS/SSL secured connections. This will be the preferred interface when used in a multi-host setup.

- Unix-domain socket driver

  This will be a POSIX local IPC socket to which the EES is listening and a peer (the PEPd) can bind.

- Named Pipe

  Very similar to the Unix-domain socket, but using a named pipe. There is less information about the peer, but has the advantage of being less platform-specific since it behaves as a regular file and hence can be used by Java.

- Direct invocation of the framework through a library call (EEF Invocation API)

  This method requires a program to load the EEF and call specific routines inside the EEF core API to initiate execution. This API is described in Section 4.1.

  This method is used by the other EICs as a 'back-end', but it is not supposed to be used directly by third-party software.

3.3. THE ATTRIBUTE AND OBLIGATION STORE

The Attribute and Obligation Store (AOS) is a 'public space' where information on attributes and obligations can be stored and retrieved during the course of handling a request. The plug-ins can use it to inspect attributes and obligations, and is able to destroy obligations which have been fully translated. After a run of the framework, the EEF uses it to construct the response message.

The AOS is initialized by the EEF. While plug-ins are initialized, they can use it to store global data. Subsequently, when the EEF is handling a request, a new thread-local AOS is created, which inherits the global data and is further filled using information passed in by the request message. For this local AOS the previously global data is read-only.

After the response message is constructed, the thread-local AOS is cleaned and the thread will terminate.

3.4. EVALUATION MANAGER

The Evaluation Manager ensures that the plug-ins are executed in the proper order. Multiple policies can be expressed in the configuration file in the Policy Description Language (PDL) [R 7] file format as used by LCMAPS. This is a human readable description of the possible state transitions in the EEF, which can take into account various fallback options in case modules do not complete successfully. The configuration file also provides the options and arguments to the plug-ins.

3.5. PLUG-IN MODULES AND MANAGER

Without plug-in modules, the EES is an empty shell. Therefore, the following basic plug-in modules are to be provided with the EES:

- Virtual machine framework

This module interacts with a virtualization framework to prepare, start, stop and clean up virtual machines.

- Scripting language interface (Perl and/or Python)

  Invoke a script using an embedded interpreter, so that a site can implement additional logic to create specific site-local execution conditions such as interaction with batch systems. The scripts will have access to the EEF API.

- Logical share to primary Unix group

  This module serves the need for basic group-based scheduling that is common to basic Torque, OpenPBS, and SGE configurations.

- Logical share to QoS/Account

  Translate the logical share to a (Maui) QoS specification or a Maui Account, both of which determine the relative priority of the scheduled task. When using the Maui scheduler, the EES can dynamically create a new QoS class for the job. It may also set a new obligation that will ensure the submitting CE will select the newly created QoS class or share for the job when it is submitted. Maui and Torque allow such dynamic manipulation through a C API, and the share can be pre-defined in the Maui configuration file.

  For other batch systems such a module can either be developed independently, or the generic scripting module can be used to implement this functionality.

- Attribute to Kerberos5 (or AFS) token

  It takes one or more attributes and obligations (usually this will be the Subject DN, but may also be the UID that was set previously by another plug-in) and procure the appropriate AFS and/or Kerberos tokens for it. These tokens are then set as obligations in the AOS and thus must be enforced by the PEP.

- NIS/LDAP directory provisioning

  Using the UID, the primary GID and the set of secondary GIDs, it ensures that the set of GIDs is properly attached to the UID in the site central directory service. This service is usually an LDAP directory using the NIS schema [R 8]. When needed, the entry will be updated. Such functionality might be required in cluster environments to ensure that the job will run on a worker node with the same set of GIDs as was used during the submission on the head node of the cluster.

3.5.1. Additional modules

Some functionality which could be provided by EES modules, and which was formerly implemented in LCMAPS plug-ins, is currently provided by the PEPd. Hence, when the EES is deployed as part of Argus, there is no strict need for these modules within the EES. On the other hand, when constructing a site-local policy, they may still be needed as building blocks. This includes the following plug-ins:

- attribute mapping to a pool UID or username
- attribute mapping to a specific pre-set UID or username
- attribute mapping to primary GID or group name
- attribute mapping to secondary GID or group name

The attributes, associated with a user, on which mappings can be based, include:

- Subject DN
- VOMS FQANS

3.6. FUTURE EXTENSIONS

3.6.1. Execution Environment life time

Execution environments established by the EES do not expire by themselves. All plug-in modules used in the EES must therefore time-stamp their information so that 'external' programs may be run (periodically) to expire unused environments. The timestamping mechanism will depend on the plug-in and the corresponding execution environment.

Hence, a possible future extension to the EES would be a management interface which can trigger plugins to clean up procured resources using this timestamped information.

Moreover, if such a management interface itself incorporates access control via the AuthZ-FW, end-entities could pre-procure, extend or release execution environments in accordance with task requirements. Management of environments would also allow for environment re-use or cloning, so that more complex or larger execution environments can be pre-staged at sites.

The design of a management interface fits well with having the EES running either as a daemon or (in conjunction with a) hosted web service, providing an interface akin to what is currently provided for the Globus Toolkit 'Dynamic Accounts Service'.

Such an interface is not specified in this document but can be defined for future versions.

3.6.2. Exclusive plugin invocation including setuid functionality

Certain policies typically cannot be executed in a multi-threaded environment. This is, for example, the case when part of a policy has to be executed under a different identity, since the behaviour of the (effective) user id switching syscalls in a multi-threaded environment depends on the details of the operating system. This could be mitigated by making use of a `setuid()` wrapper binary, like `sudo` or Apache's `suexec`.

4. EES APIS

API interfaces are provided for the EEF, AOS and plug-in components. The AOS is only exposed to plug-in components and the EEF. The Plug-in Manager and Evaluation Manager are internal compontents of the EEF, and are not externally exposed.

4.1. EEF INTERFACE

The EEF API consists of the following functions:

```
EES_RC EEF_init(char* config_file_name,
                void (*log_func)(int, const char*, va_list)
               );
```

This function initializes the EEF which will try to parse the specified configuration file and load its policy. After calling `EEF_init()`, the rest of the API can be used.

`log_func` is a pointer to a function with the signature:

```
void log_func(int priority, const char* format, va_list ap);
```

which is identical to the POSIX-specified function `vsyslog()`. The supplied log function will be accessible from the framework in the `EEF_log()` symbol.

Valid values for the internal return type `EES_RC` are `EES_SUCCESS` on success and `EES_FAILURE` on failure.

```
EES_RC EEF_startThreading(void);
```

After this function is called, the AOS will try to make use of thread-local storage instances. This means that any data that needs to be globally available must have been inserted before this point.

```
EES_RC EEF_storeRequest(char* ExecutionRequestEntity);
```

This function will try parse the SAML2-XACML2 data, whose attributes are then stored in the (possibly thread-local) AOS. The `ExecutionRequestEntity` field is a SAML2-XACML2 string as described in [R 1], which is the request message.

```
EES_RC EEF_run(void);
```

When this function is called, the EEF will try to enforce the policies specified in the configuration file until a successful mapping has been created, or all defined policies are exhausted. Obligations that have been fulfilled in this process will be removed from the AOS. New Obligations may be added which the calling party must fulfil.

```
char* EEF_retrieveResponse(void);
```

This function constructs a SAML2-XACML2 response statement using all the data present in the AOS. The return value of `EEF_retrieveResponse()` points to a SAML2-XACML2 string as described in [R 1], and is the SAML2-XACML2 response message of the EES. This function does not affect the state of the AOS.

On successful execution, the value returned by `EEF_retrieveResponse()` uses space allocated on the heap and must be freed by the calling context. When execution was unsuccessful, NULL will be returned.

```
EES_RC EEF_term(void);
```

This function terminates the EEF, cleaning up the AOS, Plugin Manager and Evalution Manager.

Furthermore two functions returning the major and minor EEF API versions will be implemented, which will always succeed:

```
unsigned int EEF_getMajorVersion(void);
unsigned int EEF_getMinorVersion(void);
```

4.2. EICS

The EES Interface consists of a number of EES Interface Components (EICs). All of these different EICs in the EI use the EEF API, as described in Section 4.1, to interact with the EEF. They will be built as separate driver modules in order to easily deal with new types of transport layer and communication protocols.

The initial version of the EES will be supplied with the driver modules, which have been introduced in Section 3.5 and will be described in the following subsections in detail.

4.2.1. Unencrypted networked invocation

This will be an HTTP server (*AF_INET* or *AF_INET6* socket). It will expect an HTTP POST of a SAML2-XACML2 request. It will add a SAML attribute with the IP adress and port information of the peer to this SAML2-XACML2 request, which will then be processed. The contents of the original SAML2-XACML2 request will not be otherwise changed or used.

The response message will be returned as a HTTP response to the caller. If the EEF will fail, a suitable SAML2-XACML2 response is formulated; in case the HTTP server will not be able to do so a suitable HTTP status code will be returned [R 8].

4.2.2. Networked invocation over TLS/SSL secure channel

This will be a HTTPS server (*AF_INET* or *AF_INET6* socket) [R 9]. It will behave in the same way as the HTTP server but using TLS/SSL secured connections in addition. The channel MUST be mutually authenticated.

The X.509 credential information of the peer will be converted into a SAML attribute and inserted in the SAML2-XACML2 request message.

4.2.3. Unix-domain socket driver

This will be a Unix-domain socket. When used within the Argus framework, it is likely that the named pipe will be used instead of this driver, since the PEPd is written in Java, which has no support for Unix-domain sockets; named pipes on the other hand behave as regular files.

When available the socket driver should get the *SO_PEERCRED* option by calling `getsockopt`(2) and insert the resulting UID, GID, and process ID information as a SAML attribute into the SAML2-XACML2 request message.

See also `unix(7)`, `socket(7)` and `socket(3)`.

4.2.4. Named pipe (FIFO)

This driver will be similar to the Unix-domain socket driver, but using named pipes instead. Access control will have to be arranged by setting the right permissions on the file descriptor. This way of communication will be available to Java applications talking to the EES and will probably be the fastest way of communication. It will, however, be difficult in multi-threaded environments.

See also `fifo(7) and mkfifo(3)`.

4.3. THE AOS INTERFACE

The AOS is a separate thread-safe subsystem within the EEF. In a multi-threaded setting each thread will see a thread-local AOS. It will have a low-level and high-level API, both available to the plug-in modules (and the EEF itself), but neither of them publicly available in the EEF-API. The preferred way of communication will be through the high-level API, which will be described here. The low-level is beyond the scope of this document and depends on the details of the AOS implementation.

4.3.1. Types

The AOS uses the opaque pointer types `aos_context_t*`, `aos_storage_t*` and `aos_attribute_t*` as handles. The type of context can be specified using the enum type `aos_context_class_t` and can take values:

- `SUBJECT`
- `ACTION`
- `RESOURCE`
- `ENVIRONMENT`
- `NONE`
- `ANY`
- `OBLIGATION`

Type `ANY` behaves as a wildcard type and can represent any of the other types.

The return value, of type `EES_RC` can either be `EES_SUCCESS` or `EES_FAILURE`.

4.3.2. Functions

Context administration:

- `aos_context_t*  AOS_createContext(aos_context_class_t);`

  Creates a new AOS context variable on the heap and returns its pointer. Returns `NULL` on failure.

- `EES_RC AOS_addContext(aos_context_t*);`

  Adds the given AOS context to the (possibly thread-local) AOS and returns `EES_SUCCESS` if successful. Returns `EES_FAILURE` on failure.

- `aos_context_t*  AOS_getNextContext(aos_context_class_t);`

  Returns the next context of specified type from the (possibly thread-local) AOS and returns its pointer. Returns `NULL` on failure.

- `EES_RC AOS_rewindContexts(aos_storage_t*);`

  Rewinds the list of AOS contexts in the given storage and returns `EES_SUCCESS` if successful. Returns `EES_FAILURE` on failure. If storage is `NULL`, the current thread-local storage will be used.

- `EES_RC AOS_destroyContext(aos_storage_t*, aos_context_t*);`

  Removes the specified context from the specified storage in the (possibly thread-local) AOS. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure. This method should be used to destroy Obligations that have been fulfilled.

- `void AOS_setContextObligation(aos_context_t*, char*);`

  Only valid for aos_context_t of type OBLIGATION. Sets the given obligation specified as char*.

- `char* AOS_getContextObligation(aos_context_t*);`

  Returns the obligation name for the given aos_context_t if this is of type `OBLIGATION`, or `NULL` otherwise.

Attribute administration:

- `aos_attribute_t* AOS_createAttribute(void);`

  Creates a new AOS attribute variable on the heap and returns its pointer. Returns `NULL` on failure.

- `EES_RC AOS_addAttribute(aos_context_t*, aos_attribute_t*);`

  Adds the given AOS attribute to the given aos_context in the (possibly thread-local) AOS. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure.

- `aos_attribute_t* AOS_getNextAttribute(aos_context_t*);`

  Returns the next attribute in the specified context from the (possibly thread-local) AOS, or `NULL` on failure.

- `EES_RC AOS_rewindAttributes(aos_context_t*);`

  Rewinds the list of AOS attributes in the given context and returns `EES_SUCCESS` if successful. Returns `EES_FAILURE` on failure.

- `EES_RC AOS_destroyAttribute(aos_context_t*,aos_attribute_t*);`

  Removes the specified attribute from the specified context in the (possibly thread-local) AOS. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure.

Attribute field setters:

- `EES_RC AOS_setAttributeId(aos_attribute_t*, char* id);`

Sets the identifier field of the given attribute to given `char*`. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure.

- `EES_RC AOS_setAttributeIssuer(aos_attribute_t*,char* issuer);`

  Sets the issuer field of the given attribute to given char*. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure.

-
  ```
  EES_RC AOS_setAttributeValue(aos_attribute_t*, void* value,
                               size_t size);
  ```

  Sets the data field of the given attribute to given char*. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure.

- `EES_RC AOS_setAttributeType(aos_attribute_t*, char* type);`

  Sets the type field of the given attribute to given char*. Returns `EES_SUCCESS` if successful, `EES_FAILURE` on failure.

Attribute field getters:

- `char* AOS_getAttributeId(aos_attribute_t*);`

  Returns the identifier field of the given attribute as `char*`, or `NULL` on failure.

- `char* AOS_getAttributeIssuer(aos_attribute_t*);`

  Returns the issuer field of the given attribute as `char*`, or `NULL` on failure .

- `char* AOS_getAttributeValueAsString(aos_attribute_t*);`

  Returns the data field of the given attribute as `char*`, or `NULL` on failure .

- `int AOS_getAttributeValueAsInt(aos_attribute_t*);`

  Returns the data field of the given attribute as `int`.

4.4. PLUG-IN MODULE INTERFACE

Modules will be loaded on-demand using the dynamic linking loader interface (`dlopen()`). Modules MUST be re-entrant, they will only see the AOS corresponding to their thread.

Compliant modules must provide the following entry points:

- `EES_PL_RC plugin_initialize(int argc, char *argv[]);`

  This function is called once within (each instance of) the EEF, and is called before any invocation of the `plugin_run()` function.This function should only do basic initialization, all the real functionality should be done by `plugin_run()`, see below. All plug-ins will be initialized before any plug-in will be run. Calls to the AOS at this stage will reference a global instance, which is shared between all plugins. The global store is only writable during this stage.

  Should return `EES_PL_SUCCESS` success, or `EES_PL_FAILURE`: failure

- `EES_PL_RC plugin_run(void);`

  Invokes the plug-in. The plug-in must process any attributes and obligations it recognises in the Attributes and Obligation Store (AOS). It should add any site-local obligations that will cause the procured environment to be selected. It must remove any obligations that have been translated into an equivalent site-local obligation. It must not remove any attributes.

  Should return `EES_PL_SUCCESS` success, or `EES_PL_FAILURE`: failure

- `EES_PL_RC plugin_terminate(void);`

  Called once before the EEF terminates. No plug-in run invocation can or will be made after this function has been called.

Should return `EES_PL_SUCCESS` success, or `EES_PL_FAILURE`: failure

Furthermore two functions returning the major and minor plug-in API versions should be implemented:

- `unsigned int plugin_getMajorVersion(void);`
- `unsigned int plugin_getMinorVersion(void);`

In addition, the following optional function will be understood by the Plug-in Manager:

- `EES_PL_RC plugin_verify(void);`

  This function is used to inform the EEF library that a run of this plug-in would leave the AOS unchanged. Should return `EES_PL_SUCCESS` if the AOS would remain untouched, or `EES_PL_FAILURE` otherwise.

5. APPENDICES

5.1. EXAMPLE PLUG-IN MODULE

```
#include "eef_plugin.h"

EES_PL_RC plugin_initialize(int argc, char* argv[]){
  /* parse options */

  /* sanity checks */

   EEF_log(LOG_INFO,  "%s:  Initialized  plugin  posix_enf  with
     options:\n", _plugin_name);
  /* log options */

  return EES_PL_SUCCESS;
}

EES_PL_RC plugin_run(){
  aos_context_t _context;
  aos_attribute_t _attribute;
  uid_t _target_uid;
  rewindContexts(NULL);
  while((_context = getNextContext(OBLIGATION, NULL))){
    if(strcmp(getContextObligationId(_context), "uidgid") == 0){
      rewindAttributes(_context);
      while((_attribute = getNextAttribute(_context))){
        if(strcmp(getAttributeId(_attribute), "posix-uid") == 0){
          _target_uid = getAttributeValueAsInt(_attribute);
        }
      }
    }
  }
  downgradeEffectiveToRealUid(&_real_uid, &_saved_uid);
  return EES_PL_SUCCESS;
}

/* terminate plugin */
EES_PL_RC plugin_terminate(){
  EEF_log(LOG_INFO, "plugin posix_enf terminated\n");
  upgradeEffectiveToRealUid(&_real_uid, &_saved_uid);
  return 0;
}
```