1 Appendix C – Performance Issues

The Authentication / Authorization solution is very complex. This might probably lead to performance issues at a later point.

Therefore in this chapter it is discussed, why certain performance-intensive parts are necessary in order to meet the requirements that have been imposed on the Authentication / Authorisation solution.

1.1 Policies are user- and object-specific

Requirements:

- It has to be possible to assign a *role* to a *user* for a specific *aggregation object* only, e.g. the user gets the role *Author* only for objects in *Admin Group A*.

- When assigning a role to a user, it has to be possible to exclude some of the *actions,* which are contained in the role, from this assignment.

Design Consequences:

- Policies have to be user- and object-specific in order to meet the required granularity. There exists one policy for each [user, action, aggregation object]-triple.

- Only then a *role* can be assigned for a specific *aggregation object* only: There are separate policies for each *aggregation object*.

- Only then certain *actions* can be excluded: Assigning a *role* to a *user* means creating a policy for each contained *action*. For the to-be-excluded *action*, no policy will be created for the user.

Performance Consequences:

- There are a big number of policies. These policies cannot be kept in memory and therefore have to be stored in the database and retrieved when they have to be checked. This retrieval costs a lot of performance, although it is slightly relieved by using a policies cache.

- Having the policies object-specific makes them very complex. Instead of having one policy for role *Author*, which the user either gets assigned or not, there are now a big number of policies to check.

1.2 Policies are too complex

Requirements:

- The requested policy functionality is very complex. Not only simple attributes of a to-be-checked object are required in order to decide on a policy, but also information that is deeply

hidden in the system. For example, for a *Content Item* not only simple values like its *status* is relevant, but also information like whether its *visibility* equals a certain *affiliation.*

Design Consequences:

- Policies are set up in a way that they require more information than is provided as input parameter of the to-be-authorized method. Therefore on each method authorization several objects have to be fetched from the system in order to decide on the permit or denial.

Performance Consequences:

- Several objects have to be fetched from the system on each request, just to decide on the authorization.

- For example, for a simple retrieval of an object, this object is retrieved twice: once for the authorization and once for the actual method invocation. Since authorization and business logic have to be separated to satisfy a clean architecture, it is not possible to use the same object in this case.

1.3 Policies-related values cannot be fetched before the authorization

Requirements:

- Depending on different roles, an action has to be checked for different kinds of objects. For example, the action *create object* has to be checked against a *Content Container* object when coming via the role *Author*, while the same action has to be checked against an *Admin Group* object when coming via the role *Administrator.*

Design Consequences:

- It would have been desireable to retrieve all data from the system that is required for deciding on authorization before the authorization engine is invoked.

- However, since the to-be-checked objects vary so much, this would mean fetching many objects from the system on every authorization request, even if in this special case only a small part of these objects would be required.

- Therefore, the required attributes cannot be fetched beforehand, but are retrieved on a one-by-one base during the authorization process.

Performance Consequences:

- The one-by-one fetching of information only works on single-attribute level due to technical reasons with the XACML engine. Even if more than one attribute is needed from the same object, a naïve implementation would fetch the object twice from the system.

- However, this problem can be partly absorbed by using an object cache, where an object is cached after the first retrieval. Still, this cache has to be cleared after each request, since the request itself or other requests might have changed the object-in-question in the meantime.

1.4 Return-value arrays have to be filtered

Requirements:

- There exist several methods in the system that have not a single object as return value but a list / array of objects instead.

- Each of these objects should only then be included in the return list, if the invoking user is authorized to access this object.

Design Consequences:

- In case a return value is a list / array of objects, these objects have to be filtered before returning them to the invoking user.

Performance Consequences:

- The XACML engine only allows one resource (i.e. target object) per XACML request. Therefore the above described filtering can only be realized as one XACML request per object in the list.

- This leads to the fact that on a method with an array of n return objects the XACML engine is invoked n+1 times: 1 time for the authorization beforehand and n times for the filtering afterwards.

- Any invocation of the XACML engine requires the generation of an XACML request from the to-be-authorized object and a matching of this request against all suitable policies in the XACML engine.

- In a naïve implementation, this would mean fetching and building all suitable policies from the database on each invocation. However, the policies cache can partly ease this problem.