Authorization Configuration

Introduction

MidPoint has a very powerful and flexible authorization mechanism. This mechanism can be used to set-up complex configurations for various occasions from enterprise delegated administration to a cloud multi-tenant environment. MidPoint starting from version 3.0 has a very fine-grained authorization model. The model uses authorization statements that can be attached to roles. The authorization model can control which parts of GUI can a user access, what operations are allowed, on which objects they are allowed and for which attributes. The authorization is enforced on several levels allowing for an unprecedented flexibility while still keeping a clean and secure access control model.

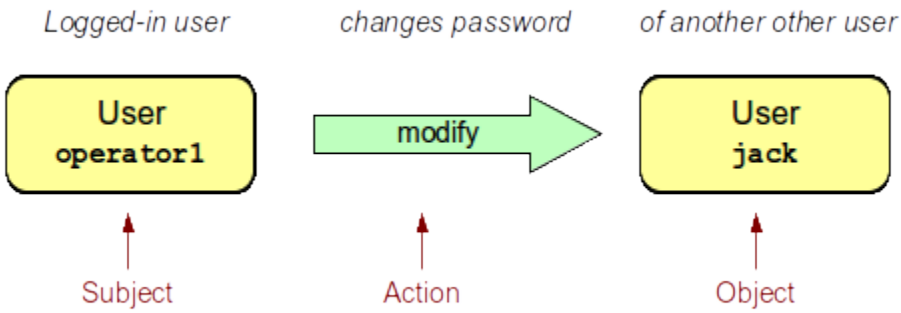
The authorization model is seamlessly integrated with the RBAC mechanism. Therefore it is easy to use even a complex authorization schemas in an IDM solution. As the authorization statements are integrated with RBAC roles then they can be hierarchical (nested), they can be subject to an approval process and so on.

Authorization Model

Authorization model of midPoint is based on the widely used subject-action-object authorization triple. Each authorization statement either contains or implies this triple. The meaning of individual elements in the triple is as follows:

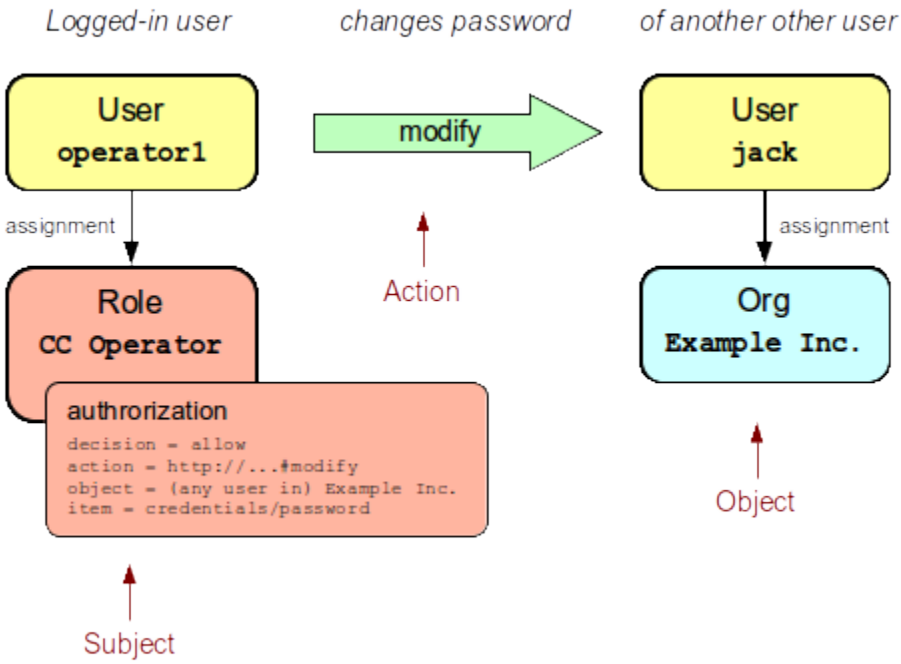
- **Subject** is the entity that initiates the action. Simply speaking it is the user that starts an action by accessing a GUI page, clicking on a button and so on. Subject is usually currently logged-in user. But it may also by the owner of a non-interactive task, client identity for a service and so on. Subject is always a midPoint User object (identified by OID).
- **Action** is the activity or operation that is initiated by subject. It can be access to a GUI page, operation to modify an object, access to a service, etc. Action is identified by an URI.
- **Object** is the entity on which the action was initiated. E.g. it can be the configuration object which is changed by saving a GUI form. Or it may be an user object whose password was changed. Object is optional. Some actions do not have object at all. E.g. an action that specified access to a "Dashboard" GUI page has no specific object that it works on. Object, if present, is always an midPoint object (identified by OID). But it can be an object of any type.

Following diagram illustrates the case when user `operator1` who is currently logged-in changes a password of user `jack`.



The basic idea of the subject-action-object approach is to create authorization statements that will either allow or deny operations for subject-action-object combinations. However it would be quite difficult to maintain large set of authorization statements if they are maintained on just one place. Therefore we have decided to integrate midPoint authorization model with the RBAC mechanism. Authorization statements are placed inside RBAC roles. These roles are then assigned to users. A user that has a role with an authorization statement is automatically considered to a **subject** of the statements. This allows very scalable and maintainable authorization set-up.

The following diagram illustrates an example of practical use of authorization statement in midPoint. It combines authorization, RBAC and organizational structure feature to create a delegated administration set-up. There is `CC Operator` role for call center operators. This role contains an authorization statement which allows modification of password (action=modify, item=credentials/password). This statement applies only to operations for which the object of the operation is a member `Example Inc.` organization. This set-up gives the call center operators to change password of any user in `Example Inc.` This is a simple example of delegated administration.



The actual XML definition of the `CC Operator` role looks like this:

```
<role>
  <name>CC Operator</name>
  <authorization>
    <decision>allow</decision>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    <object>
      <type>UserType</type>
      <orgRef oid="ce6a31e2-0054-11e4-9506-001e8c717e5b"/> <!-- This is OID of Example Inc. -->
    </object>
    <item>credentials/password</item>
  </authorization>
</role>
```

Default Deny and Other Rules

MidPoint implements **default deny** policy. What is not explicitly allowed is denied. Therefore a system without any authorization denies all operations.

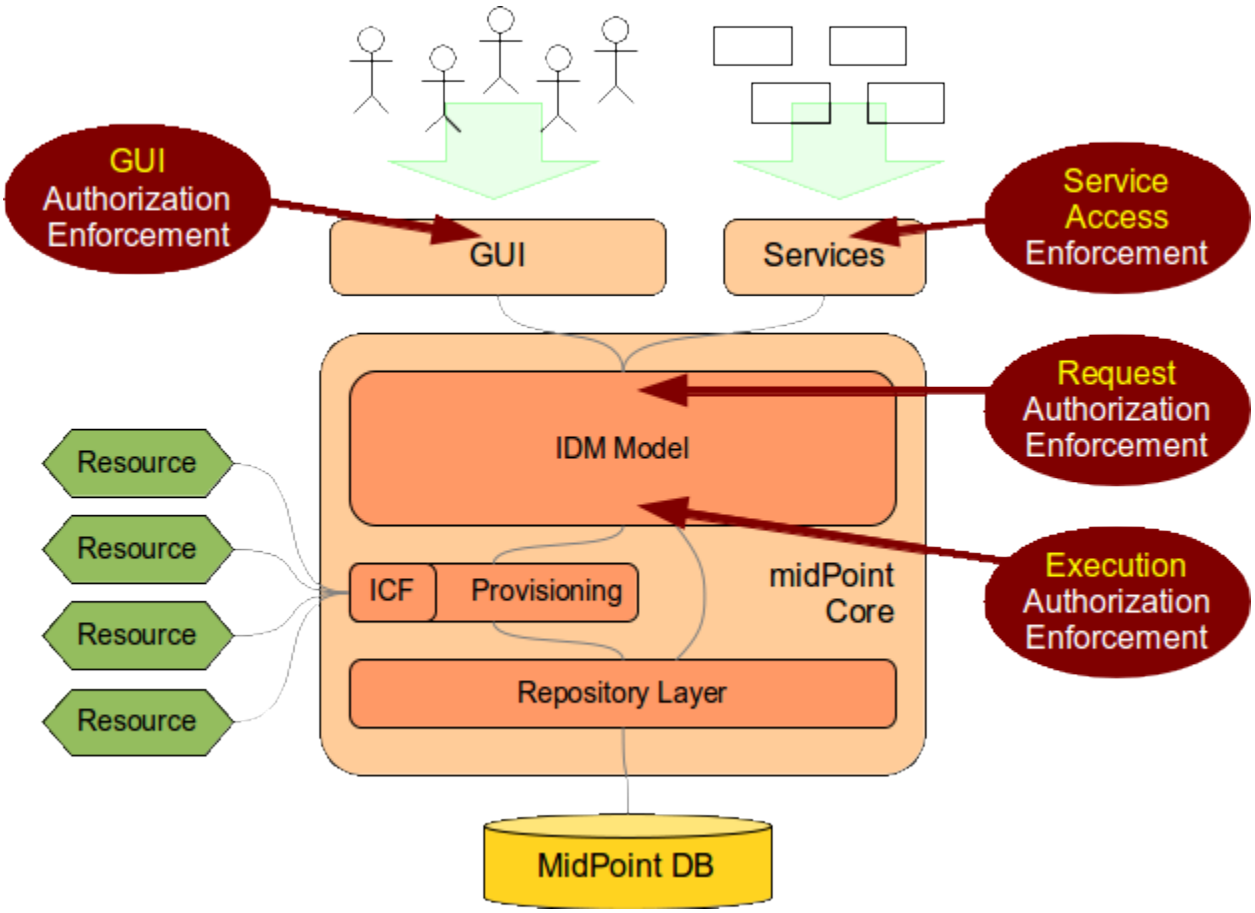
Each authorization statement can specify whether it is `allow` or `deny` statement. Each statement is first matched to the requested operation, i.e. it is checked whether subject, action and object of authorization match. It there is a match then the decision specified in the authorization statement is used. However **deny is always stronger than `allow`**. If an authorization statement denies something it cannot be allowed by any other statement. Deny decision is final and cannot be overridden. Therefore use it with caution.

If a statement does not specify any decision then `allow` decision is assumed.

Authorization Enforcement

MidPoint enforces authorizations on several layers (or phases). Each operation needs to pass all the authorization phases in order to be successful. In a normal case the operation need to pass authorization on three layers:

- **GUI** or **Service** authorization. A user needs to have appropriate GUI authorization to access the relevant GUI page. Without this authorization the operation cannot be started. This similarly applies to service access such as SOAP or REST service. An appropriate authorization to access that particular service is needed otherwise the operation will be refused as unauthorized before it even starts. GUI and service authorizations are usually very rough-grained. They grant access to a particular service or part of user interface. These authorizations usually do not have object specification and cannot be constrained to specific items. Specification of action URI is usually all that can be applied at this layer.
- **Request** authorization is evaluated for each operation of IDM Model Interface before the operation starts. This authorization checks if the user that initiates the operation has the right to request the operation in the first place. This is a fine-grained authorization. Object and item specifications can be present. This authorization is evaluated **before** the object is recomputed by the IDM model (i.e. before all the mappings and policies are applied). Properties that are result of the computation are **not** yet present in the object.
- **Execution** authorization is evaluated for each operation of IDM Model Interface **after** the object is recomputed by the IDM model and right before the operation is executed. This authorization checks if the user that initiates the operation has the right to execute the operation with all the direct and indirect effects that it might have. This is a fine-grained authorization. Object and item specifications can be present. Properties that are result of the computation are present in the object.



An operation needs to pass through all three phases to be allowed. A single `deny` in any of the phases denies entire operation. Therefore a practical authorization role should contain several authorizations to allow an operation for all the phases. E.g. the following code shows a modified version of the default End User role that is present in midPoint after a clean installation:

```
<role oid="00000000-0000-0000-0000-000000000008">
    <name>End user</name>
    <authorization> <!-- GUI authorization -->
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-3#dashboard</action>
    </authorization>
    <authorization> <!-- Request authorization -->
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read</action>
        <phase>request</phase>
        <object>
            <special>self</special>
        </object>
    </authorization>
    <authorization> <!-- Execution authorization -->
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read</action>
        <phase>execution</phase>
        <object>
            <special>self</special>
        </object>
    </authorization>
    ...
</role>
```

This role allows access to a "Dashboard" GUI page where a user can see details about himself. For this role to work three authorization statements are needed:

- GUI authorization statement for action `http://midpoint.evolveum.com/xml/ns/public/security/authorization-3#dashboard` allows access to the "Dashboard" GUI page.
- The request authorization for action `http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read` allows the Dashboard page to *request* read operation of the user object that describes currently logged-in user (defined by the `self` statement, see below).
- The execution authorization for action `http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read` allows the Dashboard page to *execute* read operation of the user object that describes currently logged-in user.

This three-phase approach may seem complex but there is a good reason for this. The details are explained below but to cut the long story short this is needed to implement a complex authorization schemes that make a fine selection of what a user can set explicitly, what can be set indirectly when a value is computed using mappings and policies and what has to be absolutely denied. However it is quite common that the same authorization statement applies to both request and execution phases. Therefore there is a syntactic short-cut. If no phase is specified in the authorization statement then the authorization is applicable to both request and execution phases. E.g:

```
    ...
    <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read</action>
        <!-- No phase specified here. Therefore it applies both to request and execution phases. -->
        <object>
            <special>self</special>
        </object>
    </authorization>
    ...
```

This is possible because the "core" authorizations work on the same actions and objects regardless whether it is a request or execution. However GUI and service authorizations use different actions and they usually do not use object specification at all. Therefore GUI and service authorization needs to be defined explicitly.

GUI and Service Authorizations

GUI and Service authorizations are usually very simple. They just contain the list of actions. Each action represents a GUI page or a service to access. E.g.

```
    <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-3#dashboard</action>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-3#myPasswords</action>
    </authorization>
```

See the GUI Authorizations page for a full list of supported GUI actions. See the Service Authorizations page for similar list of service authorizations.

"Core" Authorizations

MidPoint architecture is designed with the IDM Model component in the centre. This was designed with a purpose in mind. The IDM Model component is a brain of midPoint. It does all the policy processing, evaluates mappings, recomputes objects, projects values between objects and does all the other things of identity management logic. Placing all of this in the centre means that we can make reasonably sure that every object will be recomputed and policed as necessary. It is also an ideal place for security enforcement and auditing. And this is exactly what happens here.

Each operation is authorized when it goes through the IDM Model component. This applies to all normal operations which includes operations initiated from GUI and all the remote services (SOAP, REST) as all of these components are using the IDM Model Interface. As this interface is used almost universally in midPoint the action URIs used for authorization are also based on the operation names of the IDM Model Interface - with some minor adjustments to make them practical.

See the IDM Model Authorizations page for list of action URLs for the "core" authorizations.

Authorization phases

Each operation is actually authorized twice when it goes through the IDM Model component:

- **request phase** - when operation enters the IDM Model component
- **execution phase** - when operation leaves the IDM Model component

The important aspect to understand authorization is to understand what happens between these two authorizations. The Clockwork and Projector page explains the details. But simply speaking the object values are recomputed, mappings are evaluated and policies applied. Let's explain that using an example. Let's assume we have a user which has one LDAP account. User properties `givenName` and `familyName` are mapped to LDAP attributes `givenName` and `sn` respectively. This mapping is implemented by simple outbound mappings. If the `familyName` of a user is changed in GUI then this change is also mapped to the LDAP `sn` attribute and this is changed as well. But how about authorizations? We want to give user the ability to change the family name in the use object. This happens from time to time, e.g. when people get married. But we do not want to give the user direct access to LDAP accounts. We want to keep these account strictly controlled using midPoint policies and we do not want users to mess it up with manual changes. Luckily this is what midPoint authorization model was designed for. We need just few authorizations to implement this. Firstly the request phase authorization needs to allow user to change the `familyName` of user object. This is simple:

```
...
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    <phase>request</phase>
    <object>
        <special>self</special>
    </object>
    <item>familyName</item>
</authorization>
...
```

Secondly we need an execution phase authorization to allow this operation to be executed:

```
...
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    <phase>execution</phase>
    <object>
        <special>self</special>
    </object>
    <item>familyName</item>
</authorization>
...
```

And we also need a third authorization. Changing the `familyName` in user object will trigger the mappings and there will be yet another result: an operation to change LDAP attribute `sn`. Therefore we also need to allow this operation:

```
...
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    <phase>execution</phase>
    <object>
        <type>ShadowType</type>
        <owner>
            <special>self</special>
        </owner>
    </object>
    <item>attributes/sn</item>
</authorization>
...
```

There are several interesting things about this authorization. Firstly this is an execution phase authorization. And there is no such authorization in the request phase. This is exactly what we want. We want to allow *execution* of account modification if it is a result of policy evaluation (which means outbound mappings in this case). But we do **not** want to allow users explicitly *requesting* changes to account attributes. Therefore this authorization only allows operation in the execution phase. Secondly this authorization is using an `owner` clause to define object. This is necessary because this authorization applies to different object than previous authorizations. Previous authorizations applied to a user as an object. But this authorization applies to a shadow. It is important to realize that change of one object can result in a change of a different object, e.g. as data are mapped between focus and projections. And authorizations needs to be set up accordingly.

Object Authorization Actions

Following action URLs are used for object operations:

Operation	URL	Description	Since
Read	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read`	All read operations: getting objects, searching objects, counting objects and so on. Since midPoint 3.9 this is a short-cut for get and search authorizations (see below).	3.0
Get	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#get`	Getting objects by OID. This authorizations applies to read operations where one specific object is retrieved. Note: This authorization also applies to search results. While the search authorization governs what can be searched for and how the search filter can be specified, individual results of the search are *reduced* by using get authorization. E.g. the properties of the object for which there is no get authorization are removed.	3.9
Search	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#search`	Searching objects. This authorization applies to read operations where many objects are searched to find objects that match particular criteria. Note: Search authorization governs how the user can form a search filter and which objects are returned. But each search result is passing through additional *reduction* by using get authorization (see above).	3.9
Add	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#add`	Adding new objects. Creating entirely new object.	3.0
Modify	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify`	Modifications of existing objects.	3.0
Delete	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#delete`	Deleting objects.	3.0
Raw operation	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#rawOperation`	All operations that involve reading and changing of object in their raw representation. Simply speaking this is the XML/JSON/YAML representation of the object as is stored in the repository. Raw operations can be quite powerful as they go around all the policies. This is **not** supposed to be used in normal operation. Raw operations are intended for initial system configuration, configuration changes, emergency recovery and so on. Raw operation authorization is checked **in addition** to normal object operation. For example both `rawOperation` and `modify` authorization are needed to execute raw object modification.	3.7
Partial execution	`http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#partialExecution`	All operations that limit midPoint processing only to certain parts. This is often used to skip some parts of the processing such as approval processing, processing of certain policies and so on. Partial execution can be used to go around the policies, therefore it is considered to be a sensitive operation that requires special autorization. This authorization is checked **in addition** to normal object operation. For example both `partialExecution` and `modify` authorization are needed to execute partial object modification.	3.7

Read, Get and Search

Up until midPoint 3.9 there was only one *read* authorization that governed all the read operations. Since midPoint 3.9 there are two related, but distinct operations: *get* and *search*.

Get authorization governs operations when a single specific object is retrieved. This is usually the `getObject()` operation that retrieves objects by their object identifier (OID). This is perhaps the most frequently used operation in midPoint. It is used almost everywhere: when accounts, roles and organizational units of a specific user are retrieved, when midPoint gets information about approvers, owners, resources referenced from tasks and so on. This usually happens when midPoint follows *object references* (e.g. links).

Search authorization applies to operations that are looking through many objects. Those are `search(), searchIterative()` and `count()` operations. In this case we do not have object identifier, we are looking for an object by specifying search criteria (filter/query). Those operations are used mostly by user interface when listing objects such as users, roles and tasks. It is also applied to many operations related to organizational structure management.

In normal case both *get* and *search* authorizations are needed and in fact they are often exactly the same. But there are cases when the difference between those operations can be used to gain significant advantage. For example, it is often safe to allow get of basic properties of almost any object in the system. And this is often really needed. We want to allow users to read names of roles and organizational units that are assigned to them. We want to allows them to get information about owners and approvers of the roles that the user has access to. All of that is governed by *get* authorization. Therefore we often want to enable get for almost any object in the system (provided that only a reasonable set of properties is returned). On the other hand, we usually do not want any user to see all the other users. We want the users to see all the active employees, or all the users in their workgroup. But we do not want them to see all the archived objects. We want users to get all the roles in the system, even the deprecated or archived ones in case that they happen to still have them assigned. But we do not want those roles to appear in the searches. And this is how the difference between get and search operation can be used: give users quite a broad authorization to *get* objects. But strictly limit their *search* capability.

Possible security risk

There is a chance of system abuse in case that the users get quite a broad *get* authorization. The *get* authorization is a very simple mechanism: if OID is known, then the object is returned. The authorization does not care where the OID came from. The usual case is that the OID came from a valid object reference. But if the user learns the OID from some other channel, the user may trick the system or even abuse midPoint interfaces to gain access to an object that he should not be accessing. Therefore **it is essential not to make get authorization too broad.** Only use this approach in case when the get authorization returns reasonable and relatively harmless set of properties (e.g. only the name of the object).

The *read* authorization is still supported for compatibility and convenience reasons. It can be understood as a shortcut for specifying both *get* and *search* authorizations.

Superuser Authorization

There is one special authorization action in midPoint which can allow (or deny) any operation on any object. Following role gives a super-user powers:

```
<role oid="00000000-0000-0000-0000-000000000004" xmlns="http://midpoint.evolveum.com/xml/ns/public/common
/common-3">
 <name>Superuser</name>
 <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-3#all</action>
 </authorization>
</role>
```

The default `administrator` user in midPoint is **not** hard-coded. It is just an regular user which has the above role. This gives super-user abilities to this user. However it can be freely modified and replaced with a better least-privilege administrative model.

Object Specification

Object of the authorization can be selected in a variety of ways:

Object Selection by Type

Authorization applies only to objects of the specified type. In the following case it only applies to shadows:

```
<authorization>
  <action>...</action>
  <object>
    <type>ShadowType</type>
  </object>
</authorization>
```

Object Selection by Query Filter

Authorization applies only to objects that match specified query. In the following case it only applies to objects that have property `locality` set to value `Caribbean`.

```
<authorization>
  <action>...</action>
  <object>
    <filter>
      <q:equal>
        <q:path>locality</q:path>
        <q:value>Caribbean</q:value>
      </q:equal>
    </filter>
  </object>
</authorization>
```

Object Selection by Archetype

Authorization applies only to objects that have specified archetype.

```
<authorization>
  <action>...</action>
  <object>
    <archetypeRef oid="00000000-0000-0000-0000-000000000321"/>
  </object>
</authorization>
```

Archetype specification is multi-valued. If more than one `archetypeRef` is used in the same authorization, then *or* operation is implied. I.e. match of a single archetypes from the list will make the authorization applicable for the object.

The <archetypeRef> mechanism is available in midPoint 4.0 and later. See also Archetypes.

Object Selection by Organization Structure Membership

Authorization applies only to objects that are members of a specific Org. In the following case it only applies to member of Org identified by OID `1f82e908-0072-11e4-9532-001e8c717e5b`.

```
<authorization>
  <action>...</action>
  <object>
    <orgRef oid="1f82e908-0072-11e4-9532-001e8c717e5b"/>
  </object>
</authorization>
```

This is good for delegated administration to fixed organizational subtrees.

Object Selection by Organization Structure Relation

Authorization applies only to objects that are members of any org, for which the subject has a specific relation. E.g. this authorization type can give access to any objects that are part of any organizational unit that the subject is managing. This is illustrated in the following snippet. This authorization gives managers the ability to control any object that they are "managing".

```
<authorization>
  <action>...</action>
  <object>
    <orgRelation>
        <subjectRelation>org:manager</subjectRelation>
    </orgRelation>
  </object>
</authorization>
```

This is good for dynamic delegated administration. But please note that this authorization may degrade performance if the subject has relation to many organizational units.

The <orgRelation> mechanism is available in midPoint 3.4 and later.

Self Object Selection

Authorization applies only to objects that represent the user which initiates the operation. I.e. if the object is also a subject of the operation.

```
<authorization>
  <action>...</action>
  <object>
    <special>self</special>
  </object>
</authorization>
```

Object Selection by Owner

Authorization applies only to objects that have an owner which is specified by inner object selection.

```
<authorization>
  <action>...</action>
  <object>
    <owner>
      ... inner object selection specification goes here ...
    </owner>
  </object>
</authorization>
```

The object owner is its focal object. E.g. typical owner of account shadows is a user to whom the accounts are linked.

E.g. the following example only applies to objects that have owner who is a full-time employee:

```
<authorization>
  <action>...</action>
  <object>
    <owner>
      <filter>
        <q:equal>
          <q:path>employeeType</q:path>
          <q:value>fulltime</q:value>
        </q:equal>
      </filter>
    <owner>
  </object>
</authorization>
```

Object Selection by Tenant

MidPoint 3.9 and later

This feature is available only in midPoint 3.9 and later.

Authorization applies only to objects that have the same tenant as the subject.

```
<authorization>
  <action>...</action>
  <object>
    <tenant>
        <sameAsSubject>true</sameAsSubject>
        <includeTenantOrg>false</includeTenantOrg>
    </tenant>
  </object>
</authorization>
```

This authorization can be used to limit users to access objects only inside their own tenant. The `includeTenantOrg` element can be used to include or exclude the tenant (tenant org) itself. E.g. it can be used to prohibit modification of the tenant itself, but allow modification of any other object in its "tenancy".

This authorization works only if both subject and object are multi-tenant. I.e. it will not work if subject does not have tenant (no `tenantRef`) or in case that the object does not have tenant. Ordinary (non-tenant) authorizations should be used for those cases.

Object Selection Combinations

The object selection criteria can be combined in almost any meaningful way. E.g. the following authorization only applies to user objects that have locality set to Caribbean and are in the Org identified by OID

`1f82e908-0072-11e4-9532-001e8c717e5b.`

```
<authorization>
  <action>...</action>
  <object>
    <type>UserType</type>
    <filter>
      <q:equal>
        <q:path>locality</q:path>
        <q:value>Caribbean</q:value>
      </q:equal>
    </filter>
    <orgRef oid="1f82e908-0072-11e4-9532-001e8c717e5b"/>
  </object>
</authorization>
```

Zone of Control

MidPoint 3.9 and later

This feature is available only in midPoint 3.9 and later.

Each authorization specify *zone of control* over some part of midPoint objects. The *zone of control* is the set of objects that the authorization allows access to. Zone of control is defined by the object specification of the authorization as described above. This may be a filter, organizational structure reference and so on. If the object is part of the zone of control then the authorization is applied. So far there is nothing special about it. But it becomes really interesting in cases, when user is allowed to modify the properties that are used to set the zone of control. For example let's have a look at following authorization:

```
    <authorization>
        <name>write subtype req</name>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
        <phase>request</phase>
        <object>
            <filter>
                <q:equal>
                   <q:path>subtype</q:path>
                   <q:value>employee</q:value>
                </q:equal>
            </filter>
        </object>
        <!-- Note: subtype property is not excluded here. User could modify it ... -->
    </authorization>
```

This authorization allows a user to change the value of `subtype` property. But if the user changes the value to anything else than `employee` then such user forfeits the ability to modify this object. The object will move outside of user's zone of control. MidPoint 3.8 and earlier in fact allowed that operation. But in that case it is very difficult to set up authorization policies to make sure that the zone of control is properly maintained. The above example is very simple, but the situation may get really complicated in real-world scenarios, especially in delegated administration and multi-tenancy configurations. In such cases it was really easy to get the authorization statements wrong and give users stronger rights that intended. Therefore the behavior was changed in midPoint 3.9 and such operations are no longer allowed (but see also below). In midPoint 3.9 the zone of control is maintained. MidPoint will not allow any operation where modification of an object would result in that object getting out of authorization zone of control. This has important implications especially for multitenant deployments.

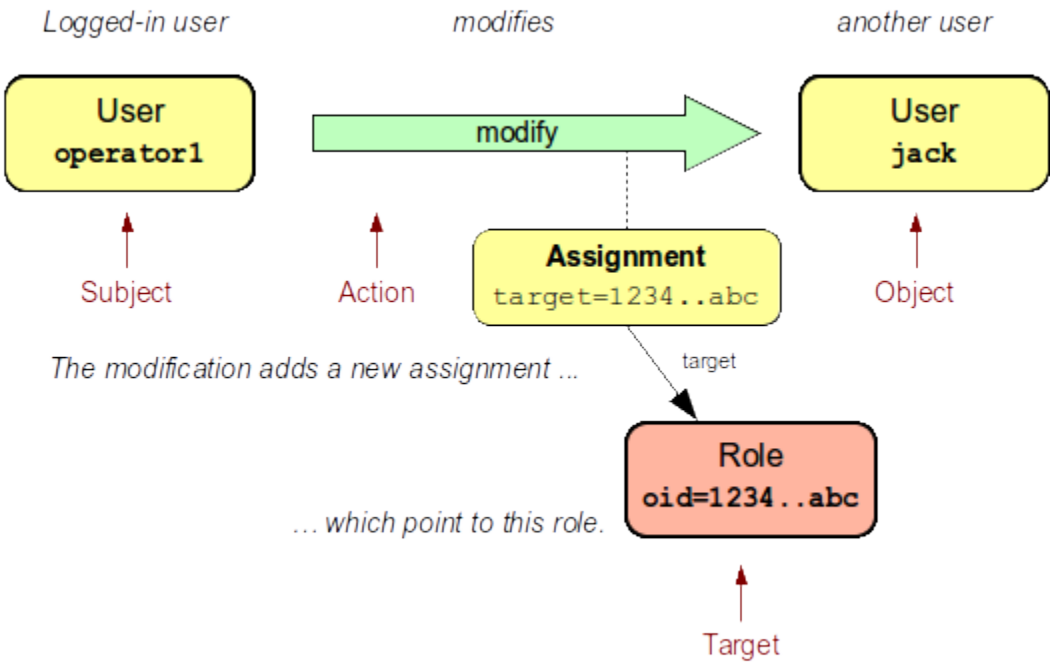
Even though the behavior of midPoint 3.9 zone of control is now more intuitive and much more secure, there may be cases when we need to allow operations that are going outside of zone of control. In that case there is a new `zoneOfControl` configuration clause for authorizations. Authorizations that need to break zone of control boundaries or authorizations that need to be compatible with midPoint 3.8 may explicitly allow such operations:

```
<authorization>
        <name>write subtype req</name>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
        ...
        <zoneOfControl>allowEscape</zoneOfControl>
        ...
    </authorization>
```

Target

Subject-action-object triple is a great model. But sometimes it is just not powerful enough. One of the common case when this model fails is complex delegated administration. E.g. if we want to give call center operator the ability to assign some selected roles to users. This cannot be achieved with pure subject-action-object model. Subject is the operator, action is `modify` and object is the user who has to get a new role. But there is no place for the role itself. Therefore the authorization mechanism based on the simple subject-action-object triple cannot deal with this situation.

Therefore the subject-action-object model needs to be extended with additional parameter: target. The target is an optional element in authorization statements that is used in authorization of operations for whose it makes sense. Assignment and un-assignment of roles and orgs is one such case. This is illustrated in the following diagram:



Therefore the target specification can be used to only select a particular group of object that can be assigned or un-assigned. E.g. the following authorization allows the assignment of application roles to any user in the organization identified by OID `1f82e908-0072-11e4-9532-001e8c717e5b`.

```
<authorization>
  <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#assign</action>
  <object>
    <type>UserType</type>
    <orgRef oid="1f82e908-0072-11e4-9532-001e8c717e5b"/>
  </object>
  <target>
    <type>RoleType</type>
    <filter>
      <q:equal>
        <q:path>roleType</q:path>
        <q:value>application</q:value>
      </q:equal>
    </filter>
  </target>
</authorization>
```

Targets can be specified using the same mechanisms as are applicable for objects (type, filter, org membeship, ...).

Assignment and Unassignment Authorizations

Assignment and unassignment are quite powerful operations in midPoint. However basic create-read-update-delete (CRUD) authorization are quite crude to address the intricacies of midPoint assignments. These authorizations can only allow all assignments or deny any assignments. There is no middle ground. And that is not very practical. Therefore there is a solution.

There are two authorizations that are designed for the purpose of controlling the assignment and unassignment on a fine level. These authorizations are designed to be target-aware. The target is the object which is assigned or unassigned (role, org, service or deputy user). This can be used to precisely control which objects may be assigned or unassigned.

However, assgin/unassign authorizations make sense only in the request phase. The primary goal of these authorizations is to limit the *targets* of assignment. And that is processed only in the request phase. All that execution phase can see is just a modification of the `assignment` container. Therefore for the assign/unassign authorizations to work correctly, you have to allow *assign* in the request phase and *modification* of `assignment` container in the execution phase. The default end user role is a good example for this.

Inducement Authorizations

MidPoint 3.9 and later

This feature is available only in midPoint 3.9 and later.

Assignment and unassignment authorization can be applied to inducements using the very same principles. There is an authorization clause `orderConstraints` that controls whether authorization applies to assignment, inducement or both.

```
        <authorization>
            <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#assign</action>
            <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#unassign</action>
            ... object, target and so on ...
            <orderConstraints>
                <orderMin>0</orderMin> <!-- order=0 means assignment -->
                <orderMax>unbounded</orderMax> <!-- order=1,2,3... means inducements -->
            </orderConstraints>
        </authorization>
```

This authorization applies both to assignments and inducements. The differentiator between assignment and inducement is so called *order*. Order of zero means assignment. Order of one or more means inducement (see Roles, Metaroles and Generic Synchronization page for more details). The `orderConstraints` clause can be used to set min/max for order therefore limiting authorization to assignment, inducements or both.

The default behavior of assignment/inducement authorizations is to apply only to assignments. Therefore if no `orderConstraints` clause is present, then the authorization allows assignments only. This behavior is slightly different than other authorization clauses, where no clause means no limitation. But this this behavior was chosen for compatibility reasons.

Expressions

This section applies to midPoint 3.7 and later.

Expressions can be used in authorization search filters:

```
    <authorization>
       <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read</action>
       <object>
           <type>RoleType</type>
           <filter>
               <q:equal>
                   <q:path>roleType</q:path>
                   <expression>
                       <!-- Make sure empty value of costCenter does not allow any access. -->
                       <queryInterpretationOfNoValue>filterNone</queryInterpretationOfNoValue>
                       <path>$subject/costCenter</path>
                   </expression>
               </q:equal>
           </filter>
       </object>
   </authorization>
```

The authorization above allows read access to all roles that have the same `roleType` as is the values of `costCenter` property of the user who is subject of the authorization.

Variable `subject` may be used in the expressions to represent authorization subject (user). Other common expressions variables may also be available or will be made available in the future. However, we recommend to avoid using the `actor` variable. Please use `subject` variable instead. Those variables are usually set to the same value. But there may be situations when the value is different (e.g. administrator evaluating authorization of a different user). The `subject` variable is usually the right one.

Authorizations are evaluated frequently. Evaluations are evaluated at least twice during ordinary midPoint operation. Authorizations are designed to be very efficient to evaluate. However, if expression is part of the evaluation then the expression may impact performance of the entire system. Expressions that use the `path` evaluator (as the one above) are usually very fast and they are safe. Even simple script expressions usually do not create any major issue. However, try to avoid placing complex or slow expressions into authorizations. Those are almost certain to have a severe negative impact on system performance. If you need complex computation, it is perhaps better to compute the value in object template and place it into property of the object (e.g. user extension property). Then use only the result of the computation stored in that property in authorization expressions.

Item Authorizations

Almost all "core" authorizations may be limited to a specific set of items. For example, read authorization may be given only to selected parts of the object by using the `item` element in the authorization:

```
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read</action>
    ...
    <item>name</item>
    <item>fullName</item>
</authorization>
```

MidPoint will adapt all its functionality to fit such authorizations. E.g. objects returned from midPoint will have only those readable fields. User interface will display input fields only for those items where the user is authorized to modify data and so on.

MidPoint 3.7 and later

Following feature is available only in midPoint 3.7.1 and later.

Item specification is a very powerful tool to implement fine-grained access control in midPoint. But with great power come great responsibilities. Which means that the authorization system is also quite complex. One of the most important details to point out is subtle but important difference between denying an operation and not allowing an operation. Authorization that denies access specifies a final decision. Denied access cannot be allowed by any other authorization. Deny authorization are very strong from a security perspective, but it is extremely difficult to combine them with other authorizations. Therefore deny authorizations are used very rarely. On the other hand if the access is not allowed by a specific authorization then it can still be allowed by another authorization. This makes authorizations "mergeable". Not allowing access is usually the right approach.

Therefore it is almost always better not to allow access than to deny access. However, enumerating all the applicable items may be daunting task if the goal it to grant access to everything except few sensitive items. There midPoint has a method for negative enumeration by using *exceptItem* element:

```
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    ...
    <exceptItem>assignment</exceptItem>
    <exceptItem>inducement</exceptItem>
</authorization>
```

This authorization grants modify access to all items except for `assignment` and `inducement`. This is still *allow* authorization, therefore it is granting access. It is not denying access. Therefore it is perfectly interoperable with other *allow* authorizations. E.g. if the user also has another authorization that grants modification of `inducement` then the system will work as expected. This also works assign/unassign authorizations.

Authorizations and Automatic Items

This section applies to midPoint 3.6.1 and later.

There are "automatic" item in midPoint that midPoint manages by itself. For example `roleMembershipRef` reference that contains a collection of direct and indirect role memberships for each focus. MidPoint will determine that automatically when assignments are evaluated. The `roleMembershipRef` values are stored in the repository so they can be used by quick search operations. There are many items like these: object and assignment metadata, role, organization and tenant references (`parentOrgRef`, `roleMembershipRef, tenantRef`), activation virtual properties (e.g. effectiveStatus) and metadata, credential metadata and many more.

Those are the items that midPoint logic controls directly. They have exception from execution-phase authorization enforcement. Their modification in execution phase is always allowed. If it was not allowed then midPoint won't be able to function properly and it may even lead to security issues.

Therefore there is a general rule: if midPoint is managed an item by itself as part of midPoint internal data management or policy management then modification of such item is implicitly allowed in the execution phase of authorization evaluation. This does not need to be allowed explicitly. However, what still needs to be allowed explicitly are the items that are modified by mappings, hooks and other customizable code. To put it simply: If midPoint modifies something by itself and there is no way to turn that off or customize it then such modification is implicitly allowed. If something is modified by a customized logic (mappings, hooks or other customization) then this is **not** allowed implicitly and you will need explicit authorization for that.

This exception applies to **execution phase only**. Request phase is not affected. All the items are still controlled by regular authorizations for request phase. Therefore these exceptions do **not** allow user to modify those items. Attempt to do so must pass through request-phase authorization first. This exception only allows midPoint logic to modify those properties without explicit authorizations.

Motivation

Strictly speaking, there would be no need for these exceptions. The modification can be allowed by regular authorizations. However, that would mean, that every practical authorization must contain those items. That is error-prone, it is a maintenance burden and it is even an obstacle for evolveability. E.g. if similar properties are added in future midPoint versions (which is likely) then all existing authorizations much be updated. The cost of slightly increased perceived security is not justified by those operational issues.

Authorizations and Performance

Authorizations are evaluated for every operations and they are typically evaluated several times. Therefore authorizations have an effect on performance. Keeping the number of authorizations to a necessary minimum is a recommendation for systems that need high performance. However provisioning systems usually prefer the ability to handle complexity over performance. And this is also the case in midPoint. Therefore midPoint still can work reasonably with a large number of authorizations if these are use with care (see the Best Practice below).

There may yet another performance consideration for authorization use. Authorizations are also used during search operations. But in this case they are used in somehow different way. When searching for an object or when listing objects MidPoint is processing the authorizations to extract a search filter from them. This filter extracted from authorizations is like a "mask" that selects only the objects that a user is authorized to see. This filter is then combined by the ordinary search filter and passed to the database for processing. This is the most efficient option. However if there is a large number of applicable authorizations and they are complex the resulting "masking" filter can be very complex. This may place additional load on the database.

Best Practice

- If possible always specify <type> in the authorizations. E.g. <type>UserType</type>. Object type is easy to determine and therefore the authorization code evaluates that first. Therefore specifying type makes the evaluation faster by quickly skipping the authorization where types do not match. This also makes the evaluation more reliable as types unambiguously determine the schema for search filters and items.
- Distribute the authorization to roles as much as possible. I.e. avoid placing all the authorization in a single role. This would mean that almost all of them have to evaluated for almost every operation. If you distribute the authorization to several roles and distribute the roles to users then a lower number of authorizations needs to be evaluated in average.

Troubleshooting

Main article: Troubleshooting Authorizations

Authorizations can be tricky. Especially if there is a large number of them and they are complex. And security best practice effectively prohibits to provide any useful error messages to the user in case that the access is denied. Therefore troubleshooting of authorization issues can be quite a demanding task - as any security engineer undoubtedly knows. However we have tried to make this task easier by implementing an authorization trace. In this mode midPoint will trace processing of all authorization statements and record that in the logfiles. The trace can be enabled by setting the following log levels:

Logger name	level	effect
com.evolveum.midpoint.security	TRACE	Enabled traces of all the security-related processing in midPoint core
com.evolveum.midpoint.security.impl.SecurityEnforcerImpl	TRACE	Enables just the processing of authorization statements and security contexts.

Please note that enabling the authorization trace has a **severe impact on system performance** as it needs to write many log records for each and every midPoint operation. This trace is not designed to be continually enabled. It is just a troubleshooting tool that is supposed to be used mostly in devel/testing environments to set up a proper security policy.

See Troubleshooting Authorizations for more details.

Examples

Self-Service Password Change

Self-service password change is one of the most widely used IDM functionality. However, the authorization setup is not trivial due to various specifics that a password has. Let's go through this scenario by starting with the simples way and ending with the right way.

The simples way how to allow change of user's own password is by using a simple authorization:

```
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    <object>
        <special>self</special>
    </object>
    <item>credentials/password</item>
</authorization>
```

This authorization will allow both request and execution of user password modification. Simple. But there are two problems.

Firstly, this authorization will only allow modification of user password. It will not allow modification of account passwords. Therefore if the user password is mapped to accounts (which is the usual case) then the operation will fail. So we need another authorization that allows modification of account password.

```
<authorization>
    <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
    <phase>execution</phase>
    <object>
        <type>ShadowType</type>
        <owner>
            <special>self</special>
        </owner>
    </object>
    <item>credentials/password</item>
</authorization>
```

This authorization allows to change password on all projections (given by `ShadowType` and `owner` combination), but only in the `execution` phase. Which means that mapped password change can be propagated. It will not allow direct change of account password. If this is desired then also `request` phase should be allowed.

The second problem with the original authorization is that there are several processes to change the password. E.g. system administrator or call center agent can change a password without specifying the old password value. This is needed to handle the case when a password is forgotten. But a normal user can change the password only if old password value is specified. Therefore there are also two different authorization setups:

- The http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify authorization represents a direct change of the password as done by system administrator. In this case the password change widget is visible in the user details form and the old value is not required
- The http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#changeCredentials authorization represents the process when user is changing its own credentials. It requires old password, proof of possession for cryptographic keys or any other reasonable safeguard. **Note:** this authorization is only applicable in the `request` phase.

Also, it is generally better to allow change of all credentials, not just password. In midPoint 3.3 and later password is the only supported credential type. But later versions will bring support for new credential types. Therefore the complete configuration for self-service password change looks like this:

```
    <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#changeCredentials<
/action>
        <phase>request</phase>
        <object>
            <special>self</special>
        </object>
        <item>credentials</item>
    </authorization>
    <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#changeCredentials<
/action>
        <phase>request</phase>
        <object>
            <type>ShadowType</type>
            <owner>
                <special>self</special>
            </owner>
        </object>
        <item>credentials</item>
    </authorization>
    <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
        <phase>execution</phase>
        <object>
            <special>self</special>
        </object>
        <item>credentials</item>
    </authorization>
    <authorization>
        <action>http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#modify</action>
        <phase>execution</phase>
        <object>
            <type>ShadowType</type>
            <owner>
                <special>self</special>
            </owner>
        </object>
        <item>credentials</item>
    </authorization>
```

Implementation note

The ...#modify and ...#changeCredentials authorizations are evaluated in almost the same way by the model. The both allow the modification of the properties specified in the `item` declaration. The primary difference is in the way how GUI presents and enforces the authorizations. The ...#modify authorization is used in the *edit schema* (refined schema). Therefore if the ...#modify authorization is present, the GUI will render a read-write widget for password. If it is not present then the password widget will not allow password change. The ...#changeCredentials authorization is not used to compute edit schema. Therefore even if it is present then the password field in the user form will still be rendered as read-only. Therefore the only way how the user can change the password is to use credentials self-service page. And this page will require old user password (if it is set up to do it).

The bottom line is that the specifics of password change interactions are implemented and enforced in the GUI. The Model is only concerned whether the password change is allowed or denied, but it does not care about the actual process.

See Also

- Troubleshooting Authorizations
- GUI Authorizations
- Service Authorizations
- IDM Model Authorizations
- Advanced Hybrid RBAC
- Organizational Structure