

Contents

Documentation	2
Current state of the implementation	2
Python API sketch	4
How to initialize a service to enable the authorization engine?	5
Enforcement in Object logic	5
Examples of a filtered search	5
Implications on UDM usage	6
Requirements for Attribute Based Access Control authorization in UDM	7
What restrictions does UDM require?	8
Integration of UDM Authoriztion using the Guardian concept	8
Permissions & Capabilities	8
Differentiation of search vs read permissions	9
Handling of UDM extensions e.g. Extended Attributes	10
Conditions provided by UDM	10
Default example roles	10
Wildcard permissions	11
Define the role of the tree structure / Comparision with LDAP ACLs	11
Known problems with Guardian	12
General/Conceptual problems	12
Ambiguity of implementation	12
Capabilities are bound to roles	12
Capabilites vs Permissions: What is API, what is opaque?	13
Restricted charset	13
Capability namespace binding restricted to the permission namespace	13
No dynamic contexts allowed	14
No removal in Guardian possible	14
Permission granting: no negative permissions	15
Permission lifecycle	15
No language to describe rules: just HTTP API endpoints with JSON payloads	16
No Caching possible: Capabilities are bound to targets	21
Guardian concepts are too abstract	21
Security Impacts	21
Availability of UDM	21
Information disclosure	22
OPA/Rego doesn't know LDAP DNs	22
Guardian as Policy Information endpoint doesn't add security to the whole system architecture automatically	23
Guardian has no way to trace decisions	23
Guardian debugability	23
Guardian API design: new/old state of targets not required for filtering	23
Performance Impacts	24
Search results must provide full data to Guardian	24
Guardian API design: extensive data format	24
UDM is synchronous	24
Rule evaluation stays on the client.	24

check-permissions vs get-permissions endpoints	24
UDM actions do a lot of sub-actions	25
Management UI Usability problems	25
Managment API bugs and issues	25
Changing conditions impossible	25
Failed decoding of input JSON data	25
Unreachable API	25
Performance of Management API	25
cyclic dependency problem	26
Questions	26
Conclusion	26

[TOC]

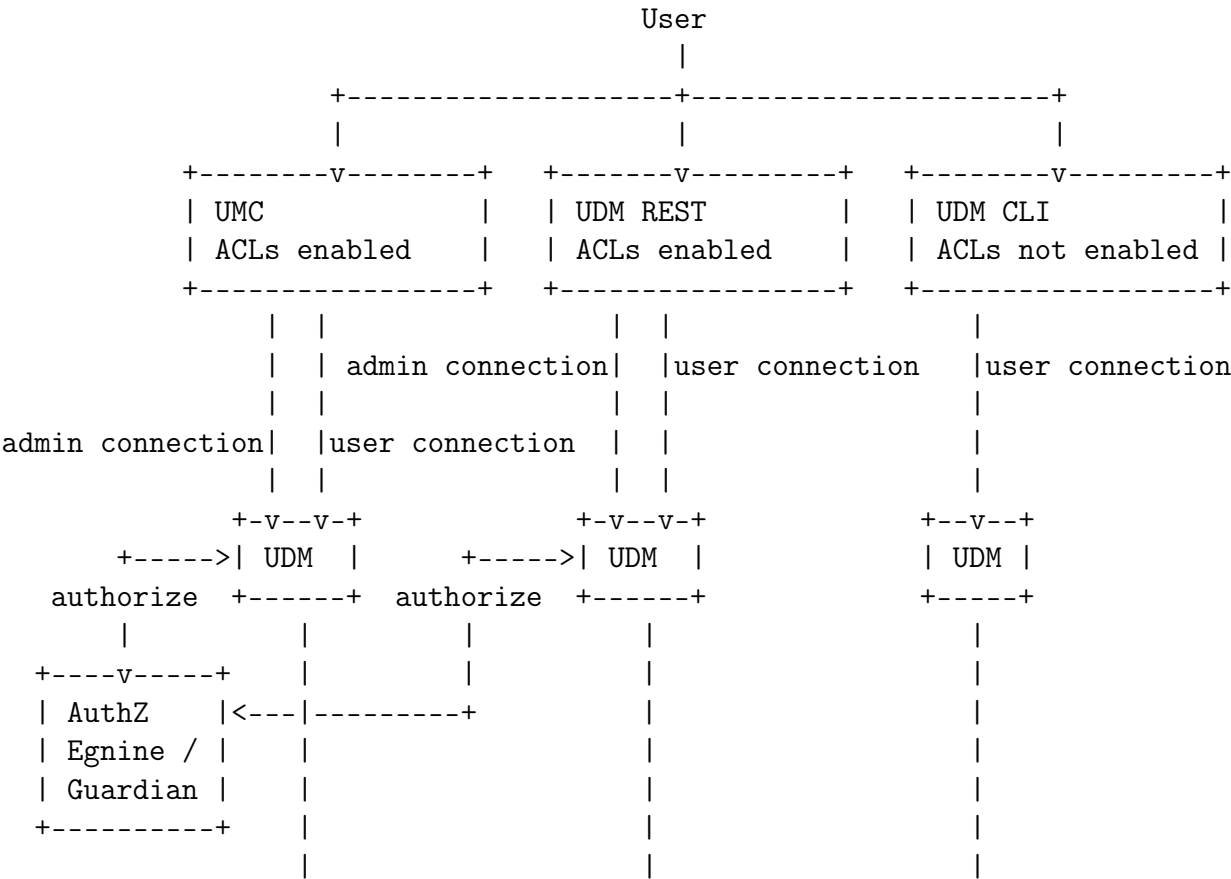
This document describes various details about the integration of ABAC (attribute based access control) in UDM. It describes the current released status, implementation details, requirements and use cases we got from Product Management and the challenged of integration Guardian and Guardian concepts into it.

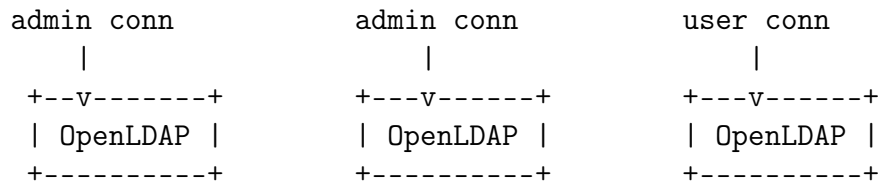
Documentation

The documentation for this feature can be obtained in ext-delegative-administration.

Current state of the implementation

Historically authorization in UDM was just realised via LDAP ACL's. The ABAC authorization concept has been integrated into UDM via Bug #58432.





The authorization checks are enabled per LDAP connection (and additionally globally per service):

It can be enabled in UMC and UDM REST API (not yet in UDM-CLI - we just didn't integrate a UCR variable for it, can possibly just be enabled there as well) via the following UCR variables:

- `ucr set directory/manager/web/delegative-administration/enabled=true`
- `ucr set directory/manager/rest/delegative-administration/enabled=true`

Via `directory/manager/rest/delegative-administration/excluded-users/.*` one can specify users which are excluded from these authorization checks, e.g. necessary so that `cn=admin` still works.

TODO: implement the same for UMC. *TODO*: implement the same for `directory/manager/rest/delegative-administration`

Configuration is still locally and not distributed somewhere. Can be configured via:

```
/usr/share/univention-directory-manager-tools/univention-configure-udm-authorization
--store-local prune /usr/share/univention-directory-manager-tools/univention-configure-udm-authorization
--store-local create-permissions /usr/share/univention-directory-manager-tools/univention-configure-udm-authorization
--store-local create-default-roles
```

A language to describe the policy rules has been implemented: a UDM domain specific language (DSL) following an extended BNF grammar. This is parsed by a LALR (Look-Ahead Left <- Right) parser.

The default rules are in: `/usr/share/univention-directory-manager-modules/udm-default-authorization`. Currently rules for the same role must be defined in the same `.policy` file. Syntax highlighting for this file (e.g. via vim) can be enabled, see `toolshed:vim/syntax/udm.vim` and `vim/ftdetect/udm.vim`

Example:

```
# Domain Administrators
access by role="udm:default-roles:domain-administrator"
  description="Domain Admins are allowed to do anything in the whole domain"
  to objecttype="*"
    grant actions="*"
    grant properties="*" permission="write"
```

Further policies can be activated via: `/usr/share/univention-directory-manager-tools/univention-configure-udm-authorization`
`--store-local create-roles --config /etc/....policy`

UDM delivers the following default roles:

- Domain Administrator (can do anything): `udm:default-roles:domain-administrator`
- Domain User (can read himself): `udm:default-roles:domain-user`
- Self Service Profile view (can write self service properties): `udm:default-roles:self-service-profile`
- Organizational Unit Administrator (can administrate users and groups in a OU): `udm:default-roles:organizational-unit-admin` (only in combination with the `udm:contexts:position`)

- context)
- Helpdesk Operator (can reset passwords): udm:default-roles:helpdesk-operator
- Linux Client Manager (can create computers/linux accounts): udm:default-roles:linux-ou-client-manager

The default roles are not attached to the corresponding groups yet.

For users with these roles, LDAP ACL's can be added which prevent granting permissions to anything. By default every user in UCS is able to read the whole contents of the LDAP directory (without sensitive data such as password hashes):

```
access to *
  by dn.base="uid=ou1-admin,cn=users,dc=example,dc=org" none stop
  by * +0 break
```

Python API sketch

univention.admin.authorization:

```
class Authorization:

    def enable(self, get_privileged_connection_callback):
        ... # enables authorization globally on the service

    def inject_ldap_connection(self, ldap_connection):
        ... # transforms user LDAP connection into connection with admin powers

    def is_create_allowed(self, obj):
        ...

    def is_modify_allowed(self, obj):
        ...

    def is_rename_allowed(self, obj):
        ...

    def is_move_allowed(self, obj, target):
        ...

    def is_remove_allowed(self, obj):
        ...

    def is_receive_allowed(self, obj):
        ...

    def filter_search_results(self, results):
        ...

    def filter_object_properties(self, obj):
        ...
```

How to initialize a service to enable the authorization engine?

```
import univention.admin.uldap
from univention.admin.authorization import Authorization

def init_the_service():
    lo_admin = univention.admin.uldap.getAdminConnection() # e.g. cache somewhere, and
    univention.admin.authorization.Authorization.enable(lambda: lo_admin)

    # get user connection
    lo = univention.admin.uldap.access(
        binddn=f'uid=oouladmin,cn=users,{ucr["ldap/base"]}',
        bindpw='univention',
        base=ucr['ldap/base']
    )
    po = position(lo.base)
    lo = univention.admin.authorization.Authorization.inject_ldap_connection(lo) # exte

def main():
    init_the_service()
    users = univention.admin.modules.get('users/user')
    univention.admin.modules.init(lo, po, users)

    user = users.object(None, lo, po, )
    user.create()
```

This injects the user LDAP connection, so that it provides `lo.authz_connection`, which can be used for certain LDAP operations. In general the regular user connection is passed to every other function call like when receiving other UDM objects.

Enforcement in Object logic

Only when doing direct LDAP operations, the admin connection has to be used, for example the `simpleLDAP.create()` method uses it like:

```
class object(simpleLDAP):

    def create(self):
        if not self.lo.authz.is_receive_allowed(self):
            raise noObject()
        ...
        if not self.lo.authz.is_create_allowed(self):
            raise permissionDenied()
        self.lo.authz_connection.add(self.dn, self.addlist)
        ...
```

Examples of a filtered search

1. using a LDAP search, searching for all attributes (assuming the results are only `users/user` objects!)

```

user_mod = modules.get('users/user')
user = user_mod.object(None, lo, po)

results = lo.search_filtered({'module': user.module}, user_filter, user_base)

2. using a LDAP search with DN's as result set (assuming the results are only users/user
   objects!)

user_mod = modules.get('users/user')
user = user_mod.object(None, lo, po)

results = lo.search_dn_filtered({'module': user.module}, user_filter, user_base)

```

Implications on UDM usage

Error handling now must be extended everywhere to catch e.g. `permissionDenied` and `noObject`. Authorization checks and object or property filtering needs to be done manually in various places, to not introduce privilege escalation or information disclosure vulnerabilities. External hooks/modules/syntax classes must implement this manually. The LDAP connection instance `lo` provides therefore the new members `authz` (providing authorization methods), `authz_connection` (being a privileged `cn=admin` connection) and other utility functions (which might change in the future).

The use of `authz_connection` can be compared with encoding/decoding of data. The `lo.authz_connection` must be used for all operations like `get_schema()`, `add()`, `modify()`, `rename()`, `delete()`, `getPolicies()`, `get()`, `getAttr()`, `search()` and `searchDn()`. The `lo` connection must be used for all other operations, e.g. passing to UDM objects `univention.admin.modules.lookup(lo, ...)`, `univention.admin.modules.get("users/user").object(lo, ...)` or `univention.admin.modules.init(module, lo, ...)`

Additionally the methods `get()`, `getAttr()`, `search()` and `searchDn()` must be guarded, to filter the results for information the user is not allowed to read! E.g. alternatives are `lo.search_filtered()` and `lo.search_dn_filtered()`.

In general it's very important that things raise the same exception signature like LDAP would do, if no permissions exists, otherwise with that information leak it would be possible to find out which objects exists, especially if the user has control over the used LDAP filter, it can be used to obtain arbitrary domain data.

The implementation still has several possible vulnerabilities:

- probably not all endpoints of UMC and UDM REST API are covered
- various endpoints which offer the possibility to provide an LDAP filter or LDAP search base + scope, which doesn't respond equally to if the object doesn't exist in LDAP
- information disclosure in various methods where the behavior differs from LDAP. most known places we e.g. raise `noObject` instead of `permissionDenied`. leftover: whole tree hierarchy must be checked.
- information disclosure via specially crafted LDAP DN's might allow this (e.g. `uid = Administrator` is equal to `uid=Administrator`). This is currently handled safely. But if we switch to real Guardian and evaluate this with rego, it will probably be vulnerable.
- time based side channel attacks might be possible now, and guaranteed possible if there is HTTP communication involved.
- UDM modules which implement `lookup()` on their own, instead of using the generic variant e.g. `computers/ipmanagedclient`.

- DNS and DHCP objects if access to computers is allowed I don't consider those information secret!? They could also be fetched via a DNS request.
- UDM modules with a defect `hasChange()` method
- pseudo-properties without LDAP attribute, might be not correctly detected by `hasChange()`
- UDM objects which aren't `open()`ed, when the open logic adds state
- It's possible to "touch" any object, by sending a modify request without changed properties.
- All marked places in the UDM source code matching: `git grep 'information disclosure'`
- probably many other things

From product management we were not allowed to implement this feature by creating mappings to LDAP ACLs, which would make us safe against all this (and would have been a lot easier to implement). Therefore we have to address them somehow.

Other vulnerabilities are possible in the rules itself. Creating rules requires knowledge about the object structure. To protect against privilege escalation we must prevent write access to (certain values of) properties like `users/user` or `groups/group:password,serviceSpecificPassword,guardianRoles,guardian`. Also certain paths in `homeSharePath` and other attacks are imaginable.

Other wrong role definitions can lead to information loss, e.g.: If the actor user has access to read group XYZ and another actor modifier of groups has not, the groups will be removed.

The place of implementation is also kind of wrong: Authorization checks are done very early in `modify()` / `create()` / etc to prevent that the `_ldap_pre_create()` / `_ldap_pre_modify()` logic, which already does things like moving / creation / modification of referenced objects, can be executed by any arbitrary actor, not having the required permissions. But exactly that logic often sets default values, modifies states, executes hooks, etc which is then not covered by the authorization checks. Maybe it would be best to just check the permissions twice.

Requirements for Attribute Based Access Control authorization in UDM

- The use of LDAP ACL's was declined by the Product Management (**strategic decision**).
- Authorization for UDM based on objects types and properties, not LDAP attributes **strategic decision**
- Authorization based on
 - the role (`guardianRoles`) of the *actor*
 - the DN (`position`) of the *target*
- Probably a future need for "value-based" authorization (e.g. `username` can not be `root`, group memberships can not be `Domain Admins`, a set of certain `guardianRoles` are allowed/declined)
- Privileged LDAP connection for access to database after authorization (not a user connection)
 - This introduces a security risk for UDM extensions. We can not technically enforce that Python code from 3rd parties do the authorization part.
 - * **Product Management decision:** we can ignore this for now, it is the responsibility of the operator and/or the "manufacturer" to ensure that 3rd party extensions work correctly
 - * Furthermore we want to provide the possibility to extend UDM in a declarative way (e.g. via `YAML`) to get rid of custom Python business logic

- Customers can create roles and assign permissions for those roles
- If possible, the Guardian should be used

The requirements and use cases are still vague and have to be further clarified. We need to support three roles: OU Administrator, Computer Join Operator, Helpdesk Operator.

Certain knowledge about the product helps to implement this:

- The security implications of certain attributes
- The current UCS LDAP ACL's
- The LDAP ACLs of UCS@school
- The requirements of the self service

With this in mind (and considered), we know that the product should go into the direction that these same things can be realized with UDM permissions.

What restrictions does UDM require?

- no writing to `users/user:guardianRole`, `groups/group:guardianMemberRole`
- no reading of `users/user:password`, `users/user:serviceSpecificPassword` (but maybe write)
- sensitive rules about `users/user: overridePWLength`, `overridePWHistory`, `shell`, `unixhome`, `locked/unlock`, `disabled`, `gidNumber` `uidNumber`.
- restrict the possible values for `primaryGroup` and `groups` (don't allow to put into Domain Admins)
- ... to be continued ...

Integration of UDM Authoriztion using the Guardian concept

Guardian is an Authorization Information Point, meaning it doesn't enforce any policies. UDM is the layer which enforces the rule evaluation.

Currently Guardian has many drawbacks, so that we implemented layers which abstract away Guardian concepts and implementation details. Those are now further explained.

Permissions & Capabilities

UDM by default provides permissions:

- A namespace for each UDM module: `udm:{module}:`
- Capabilities for general actions of a all UDM modules (if the module supports the action):
 - Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:create` allows to create objects of this module
 - Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:modify` allows to modify objects of this module
 - Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:rename` allows to rename objects of this module
 - Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:remove` allows to remove objects of this module

- Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:move` allows to move objects of this module
- Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:search` allows to search objects of this module
- Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:read` allows to read a specific object of this module
- Capability: `condition objectType == "users/user"` grants the permission `udm:{module}:report-create` allows to create a report for objects of this module
- For every UDM property of all UDM modules:
 - Permission `udm:{module}:read-property-{property}` allows to read the property value
 - Permission `udm:{module}:search-property-{property}` allows to use the property in a search filter
 - Permission `udm:{module}:write-property-{property}` allows to write the property value in a `create` or `modify` action
 - Permission `udm:{module}:readonly-property-{property}` restricts the `write` permissions back to `read` permissions again
 - Permission `udm:{module}:writeonly-property-{property}` restricts the permission to `write` without having `read` or `search` permissions
 - Permission `udm:{module}:none-property-{property}` restricts the `write` or `read` permissions back to `none` permissions again
- Wildcard-Permissions:
 - For the realization of a simple Domain Administrator role, wildcard permissions exists. They MUST NOT be used otherwise.
 - `~~Permission: udm:{module}:*` allows any of the above UDM module actions~~ (overcomplicates Guardian handling)
 - Each of the per-property permissions above allows to specify `*` as property name, which applies to all properties the module provides.
 - This is security critical and then must be combined with further restrictions, for example:
 - * `udm:{module}:write-property-*` +
 - * `udm:{module}:readonly-property-guardianRoles` +
 - * `udm:{module}:none-property-password` or `udm:{module}:writeonly-property-password`

Differentiation of search vs read permissions

What would it help to differentiate a **search** permission, while we already have a general **read** permission to retrieve a specific single object?

1. UDM modules support a optional **search** operation. If no permissions for searching exists for the specific module, the search form is not rendered (but opening specific objects is possible e.g. via the LDAP directory tree). The same applies to the **report-create** permission, which makes the button in the UI (in)visible.
2. Another use case could be, that users should not get the permissions to open a certain object type, while the user object itself allows to choose objects of such objects in a selection list. Filling the selection list then requires the permission for the **search** operation.
3. We could differentiate wheather a property can not be used in a search filter, e.g. prohibit searching for `password=*`, `fetchmailPassword=*`, etc.

Handling of UDM extensions e.g. Extended Attributes

After the installation of an UDM Extended Attribute, or an app which provides UDM Modules or properties, new **permissions** and **capabilities** must be registered in Guardian. It might be necessary to adjust default roles. Especially regarding wildcard permissions, which are partly resolved into a lookup of all modules and properties, require the adjustments of default roles e.g. so that the Domain Administrator is still allowed to act on every object.

The rule re-creation can currently be achieved via: `/usr/share/univention-directory-manager-tools/u"$@" create-permissions`

Conditions provided by UDM

Permissions are granted by Capabilities, which can restrict the permissions by adding certain Conditions.

UDM provides the following conditions:

- `udm:conditions:target_position_in` with parameters `position=`, `scope=` which restricts the permissions to the given LDAP DN position using one of the LDAP scopes `one`, `base`, `sub`.
- `udm:conditions:target_position_from_context` with parameters `context=`, `scope=` which restricts the permissions to the LDAP DN position read from the given context using one of the LDAP scopes `one`, `base`, `sub`.
- `udm:conditions:objecttype_equals` with parameters `objectType=` which restricts the action to any UDM modules name.

Not fully implemented / untested: (

- `udm:conditions:target_property_value_compares` with parameters `property` `operator` `value`, where operator is one of: `==`, `!=`, `regex-match`, `regex-nomatch`, `==-i`, `!= -i`, `regex-match-i`, `regex-nomatch-i`
- `udm:conditions:target_property_value_dn_compares` with parameters `property` `operator` `value`, where operator is one of: `==`, `!=`, `==-i`, `!= -i`, `subtree`, `onelevel`, `subtree-i`, `onelevel-i`.
- `udm:conditions:property_equals` with parameters `property=` which restricts the action to any UDM modules property name. Must be used with the AND operator and the previous condition.
- `udm:conditions:action_equals` with parameters `action=` which restricts the action to any of `create`, `modify`, `rename`, `remove`, `move`, `search`, `read`, etc.

)

Default example roles

UDM provides some default roles, which are using some builtin capabilities:

- `udm:default-roles:domain-user` allows to read everything non-sensible
- `udm:default-roles:domain-administrator` allows to write everything
- `udm:default-roles:organizational-unit-admin` allows to be admin of a certain OU given by its context
- `udm:default-roles:helpdesk-operator` only allowed to set passwords of users underneath of a certain OU given by its context

- `udm:default-roles:computer-join-administrator` allows to join computers into the domain, and set their corresponding password
- `udm:default-roles:self-service-profile` allows to modify the properties of the own user object specified in UCR variable `self-service/udm_attributes`

Wildcard permissions

Wildcard permissions allow to not be required to specify a whitelist of all allowed properties to have access to. UDM is a dynamically extensible framework, where new properties can be added 1) via additional schema extensions and 2) on the fly via UDM “extended attributes”.

Supporting wildcard permissions adds a security risk. If customers need to copy our default capability definitions for an example role and in some errata update we introduce new security-relevant properties e.g. another service specific password, customers need to adjust their access definitions prior to the software upgrade. Otherwise a vulnerability time window exists, where access is allowed to these attributes, even if the customer references (in a currently impossible way) our default policies.

Alternative concept to wildcard permissions would be something like a “permission bundle”, where we group certain properties, e.g. `default-users-user-properties`, `custom-users-user-properties`, `sensitive-users-user-properties`.

The conflicting issue: We have to offer security by default and usability and maintainability of rules for the customers.

Define the role of the tree structure / Comparison with LDAP ACLs

In the whole concept we ignore that a directory service is a tree-based directory. In LDAP, in order to be able to read an object, you need `read` or `search` rights to the `entry` attribute on all parent objects and if you want to modify the object, you need `read` rights to the `children` attribute on the parent object.

Example: to find and `read uid=foo,cn=bar,cn=baz,cn=users,dc=example,dc=org` at least the following access rights are required:

- `read` on `uid=foo,cn=bar,cn=baz,cn=users,dc=example,dc=org` `attr=entry, objectClass, etc`
- `search` on `cn=bar,cn=baz,cn=users,dc=example,dc=org` `attr=entry` (wenn man hier z. B. `add` auf ein Kindobjekt machen will, dann auch `write` auf `children`)
- `search` on `cn=baz,cn=users,dc=example,dc=org` `attr=entry`
- `search` on `cn=users,dc=example,dc=org` `attr=entry`
- `search` on `dc=example,dc=org` `attr=entry`

Or you need to know exactly the object DN, then you can read it if you have the corresponding permissions on the object and parent. But how do we want to realize the LDAP directory tree module, if you cannot see intermediate objects? Leave it to the customer to create all the permissions?

A regular LDAP search with the `base=dc=example,dc=org` for all objects with additional Guardian filtering will yield results where the inter-between objects might be missing. E.g. it will yield `cn=baz,cn=users,dc=example,dc=org` and `uid=foo,cn=bar,cn=baz,cn=users,dc=example,dc=org` but not `cn=bar,cn=baz,cn=users,dc=example,dc=org`.

Shouldn't this behave like LDAP?

Known problems with Guardian

The Guardian component currently has several drawbacks, which has to be solved or circumvented, so that UDM is able to realize a authorization concept by using Guardian.

General/Conceptual problems

Ambiguity of implementation

“There should be one – and preferably only one – obvious way to do it.” Zen of Python

Guardian allows several ways to implement a use case and its documentation doesn't give clear answers how things are supposed to be solved.

One can implement a:

- `udm:udm:users-user-{action}` permission, where action is e.g. one of `create`, `modify`, `remove`, `rename`.
- `udm:udm:{action}` permission, which must be used in combination with a condition, checking the `objectType == 'users/user'`
- `udm:udm:action` permission, which checks in multiple conditions that `action == 'modify'` and `objectType == 'users/user'`

One can implement a:

- `udm:udm:users-user-write-property-description` permission
- `udm:udm:write-property-description` permission, which must be used in combination with a condition, checking the `objectType == 'users/user'` and `property == 'description'`
- `udm:udm:action` permission, which checks in multiple conditions that `action == 'modify'` and `objectType == 'users/user'` and `property == 'description'`

One can implement a:

- Portal Tile view based on (virtual) roles present in the target
- Portal Tile view based on the app name in the permission string

Change request: The Guardian manual should clearly state in which way things should be implemented and tell it's possible advantages and disadvantages.

Capabilities are bound to roles

When we want to implement customer use cases in a generic reusable fashion, by providing examples how to realize them, we would create explicit permissions (e.g. `udm:udm:users-user-modify-property`). That would serve as a clear API and allows easy re-use by a customer. If we want to implement generic capabilities, which would serve instead as the API, customers need to copy the whole structure and if we change something in our required examples, they need to adopt it equally to their copies. Capabilities are bound to a specific role. That's a problem and doesn't allow re-use but requires hard-copying. To be useable, a capability definition should stand on its own, not yet assigned to any role. Maybe some concept like a capability bundle could enhance this.

Change request: The Guardian should remove the role assignment from the capability and introduce a different layer in the Management API to link (multiple) capabilities to (multiple) roles.

Capabilities vs Permissions: What is API, what is opaque?

Guardian introduces all these different concepts but doesn't clearly state, which objects are supposed to be created by an App and which are supposed to be created by an administrator. The Management UI just shows all of them.

Permissions can grant something, while in practical use, they often require specific conditions to be bound to them. E.g. a `app:actions:create` permission must be used in combination with a condition `if type == "teacher"`. This permission with a condition builds a capability to create a teacher.

Of course there could be a permission `app:teacher:create`, but at least UCS@school doesn't use it this way and discourages to use it like this. But Guardian currently has the drawback, that as mentioned above, capabilities are bound to roles. So, by design, they are not re-useable and lead to copy & paste, which then makes them not-evolvable.

Customers should not get the cognitive load to understand both concepts. We could learn from FreeIPA how to design this very simple (KISS principle), they just differentiate between role, permission and privilege. See Free IPA Permissions.

It doesn't feel correct, that permissions create a API. Managing single permissions to a UDM properties can't evolve. Only pre-defined permission sets by use can do this.

Change request: Guardian should throw away the concept of capabilities and provide *Privileges*. which are the external API and referenced by customers. Apps provide these privileges. Permissions should be nearly opaque - implementation details which might be changed by the app at any time.

Restricted charset

Guardian restricts the charset of permissions (basically: `[a-z0-9]`). UDM modules, properties and extended attributes allow arbitrary characters. We need to create a mapping of UDM module and UDM property names.

- The mapping is irreversible when we need to strip characters (Increases the complexity very much in handling this).
- The mapping is error prone
- For Administrators this is also intransparent, as they have to remember `pwdChangeNextLogin` and `pwdchangenextlogin`.

1. It's not possible to use `users/user` or `pwdChangeNextLogin`.
2. It would be nice to use the `:` as separator for concepts in permission names e.g. `{app}:{namespace}:$module:$action:$thing`.
3. It would be nice if the whole character set of LDAP DN's is allowed (see later about "Contexts").

Workaround: lowercase all values and replace special chars with `-` and nothing

Change request: The Guardian should allow for app names, namespaces, permissions and in contexts all printable ASCII characters, except for the separator `&`.

Capability namespace binding restricted to the permission namespace

Guardian has a concept of `app:namespace:thing` for permissions, capabilities, roles, etc. In UDM it would have been nice to use the namespaces for the UDM modules. e.g.: `udm:users/user:...` would allow things like `udm:users/user:{action}:{property}:{details}`.

But capabilities are bound to be in the same namespace as its permissions. So it is not possible to create a capability which grants permissions for more than one UDM module.

Workaround: Create a capability for each namespace (UDM module), so that a domain administrator e.g. needs to get over 100 capabilities assigned.

Change request: The Guardian should allow **capabilities** - to reference **permissions** outside of the **namespace** the capability is created in. - to be cloned, so that they can be modified more easily interactively. - to be inherited from another **capability**, and then the **sub-capability** can add further **conditions** (linked with any relation) which are linked with the **super-capability** via **AND**.

The problem realizing this, is that the Guardian Authorization Engine cannot just search for all capabilities in a certain namespace anymore but must inspect all capabilities, which include permissions of a given namespace.

No dynamic contexts allowed

While we would benefit from dynamic permissions, because UDM properties and modules can be added dynamically, having such functionality is crucial for **Contexts**. If we want to use the **context** Guardian concept we want to extend permissions so operations are e.g. restricted to targets underneath of a certain OU.

1. A context cannot have the full character set of a LDAP DN (basically also restricted to [a-z0-9]). So we cannot add a context `udm:udm:ou-admin & udm:contexts:ou=ou2,dc=example,dc=org`
2. Every context must be registered in guardian. This requires that everytime a OU is created or removed, the context must be added/removed in Guardian. A listener module would have to do this, which adds unnecessary overall complexity to the whole system.

A concept of named context would be good, so that the value is freely chosen but the context name is registered in guardian. One could name a context `udm:contexts:ou` and its value would be `ou=My school 1,dc=example,dc=org` so that the resulting role string would be `udm:udm:organizational-unit-administrator&udm:contexts:ou=ou=My school 1,dc=example,dc=org`.

It would allow be nice if multiple contexts could be given per role, not just only one. This could be realized by separating them by `&`.

→ Values for contexts should be free-form strings, without having to be registered in Guardian.

Workaround: Use the context name, rewrite the roles to strip out parts of the context and gather the data hardcoded in UDM code and provide them via Guardians **extra-arguments**

Change request: The Guardian should allow to register contexts and allows assigning free-form string values for them in the role string by separating them via the first = (e.g. `udm:default-roles:organizational-unit-administrator&udm:general:ou=ou=My school 1,dc=example,dc=org`).

Change request: The Guardian should allow to specify multiple contexts in a role string, separated via `&` (e.g. `udm:roles:foo-role&context1&context2`).

No removal in Guardian possible

The guardian management API and UI doesn't allow to remove any created object like **app**, **namespace**, **role**, **context**, **permission**, **condition**. Only a **capability** can be removed.

Change request: The Guardian Management API should implement the endpoints to remove Guardian objects (and the UI should use it). Fix `univention/dev/projects/authorization-engine/guardian#262`

Permission granting: no negative permissions

Guardian allows to give permissions but has no way to reduce already given permissions by a further capability (additive permission management without subtractive permission management).

This is fundamental for our requirements. UDM is dynamically extensible and is also evolving, so that we sometimes add new properties.

1. Guardian requires for each changeable property a permission to be defined.
2. Guardian doesn't have a wildcard permission granting concept
3. If we introduce a wildcard concept, we cannot use the check-permission endpoint anymore: We have to do the whole rule evaluation in UDM.
4. If we introduce a permission-restriction-afterwards concept we cannot use the check-permission endpoint anymore: We have to do the whole rule evaluation in UDM, see the below realization idea how Guardian could support it.
5. If we don't have a wildcard concept, we have to adjust existing rules on every software upgrade / extended attribute.
6. If we don't have a wildcard concept, we will send very large amount of permissions strings on each rule evaluation check.

Drawbacks of negative permissions:

- If a user contains two roles, e.g. `helpdesk-operator` which grants access to `users/user:write-property-password` and another OU specific role which disallows access via `users/user:none-property-password`, the functionality of the first role would be destroyed: In some cases this could mean that you have less permissions then before, which is not something Administrators would expect. This can partly be solved by adding specific conditions to the capabilities which let the permissions only apply to e.g. a certain subtree.

Implementation idea: Guardian could also provide parameters to check for the absence of permissions.

```
permission_check(  
    general_permission=["users-user:modify"],  
    target_permission=["users-user:write:description"],  
    not_target_permissions=["users-user:readonly:description"],  
    not_permissions=["users-user:none:description"]  
)
```

Permission lifecycle

As pointed out before, UDM is a evolving system over time. From time to time, we add new UDM properties or new UDM modules. Or customers do it via extended attributes. Or apps do it via their joinscript / app installation. We most recently had the problem, that the existing LDAP ACL's prevented us from doing certain changes in UDM regarding the `univentionObjectIdentifier`.

UCS development takes place in erratum-updates, even for new features. We don't have a

way to signal customers that they need to change their existing role set, after we added a new property. Not having the permission to write this attribute will cause the object creation to fail. UDM REST API and UMC require the full data representation of the new object to be send. Customers will also not see the reason for it. The error message will just be **permission denied** without the information which attributes are not writeable. Guardian will not provide this information. Maybe when looking up the log files. This should not end in a support case. UCS cannot change the defined roles of customers.

Guardian has no concept for the permission lifecycle.

Implementation idea: If Guardian would provide a feature for re-useable sets of permission and capability bundles (privileges), which serve as an API, UDM could provide useful defaults.

No language to describe rules: just HTTP API endpoints with JSON payloads

We expect that customers don't just want to define their policies by clicking them together in the Guardian Management UI but to roll out a deterministic set of rules. This requires a data format. JSON is (usually) not a data format expected to be written by humans/administrators but generated by software. YAML is more common to be written by Dev-OPs teams. Administrators probably also don't want to write HTTP clients (which then must be maintained as well) to push policy rules into the system.

My (@fbest) personal opinion is, that realizing the Guardian configuration via the Guardian Management API as a HTTP service is a misconception. It adds a new custom specific JSON data format, another Univention-Invention, which must be learned and implemented by clients/ISVs/customers. We even don't achieve to make use of the power of OpenPolicyAgent.

Guardian defines the semantic of a language by introducing concepts like "permissions", "capabilities", "conditions", "roles", "contexts", etc. Guardian provides no real syntax to realize these concepts. Instead it offers unfinished JSON based HTTP API endpoints with (over-)complex data structure. See also [Guardian API design: extensive data format](#).

Other access control systems make their rules describable via a language, such as YAML, URI, ACI (objects in the directory Active Directory), LDIF or custom DSLs (e.g. LDAP ACLs). Guardian allows to describe rules as capabilities, where each capability is like the following JSON format POSTed to the `guardian-management-api`.

A capability grants permissions to a specific role based on optional conditions:

```
{
  "name": "groups-group-creation", # must be in the same namespace as the permissions i
  "display_name": "Allow the role udm:udm:groups-group-creator to create UDM groups/grou
  "role": {
    "app_name": "udm",
    "namespace_name": "roles",
    "name": "groups-group-creator"
  },
  "relation": "AND", # operator in which conditions are linked
  "conditions": [
    {
      "app_name": "udm",
      "namespace_name": "conditions",
      "name": "objecttype-equals",
      "parameters": [
```

```

        {"name": "objectType", "value": "groups/group"}
    ],
    {
        "app_name": "udm",
        "namespace_name": "conditions",
        "name": "position-in",
        "parameters": [
            {"position": "cn=groups,dc=example,dc=com", "scope": "subtree"}
        ]
    },
    {
        "app_name": "udm",
        "namespace_name": "conditions",
        "name": "action-equals",
        "parameters": [
            {"name": "action", "value": "create"}
        ]
    }
],
"permissions": [
    {
        "app_name": "udm",
        "namespace_name": "udm",
        "name": "create"
    },
    {
        "app_name": "udm",
        "namespace_name": "udm",
        "name": "write-property-name"
    },
    ...
]
}

```

This data format is bloated, not changeable, not directly focused on what it offers, not intuitively understandable. Changing the underlying implementation will change everything.

UDM should abstract away these implementation details and provide a easy to understand format, where this HTTP API JSON syntax is created from.

We assume customers don't want to write such error prone JSON blobs. We should offer configuration formats and ways which are easy to write, have no redundancy, focus on being intuitively understandable, align to known concepts. So what are configuration formats which are nearly standards, and easy to parse?

Example: One example to describe the default rules for our uses cases is the following YAML, which are 100 lines of YAML instead of (the current) 24.362 lines of JSON within 143 files. This is very specific to the terminology and concepts of Guardian. It could be used a a general purpose configuration format for Guardian.

```

conditions:  # named reusable conditions
users-user-objects:

```

```

    udm:conditions:target_object_type_equals:
      objectType: "users/user"
groups-group-objects:
  udm:conditions:target_object_type_equals:
    objectType: "groups/group"
object-access-underneath-own-ou:
  udm:conditions:target_position_from_context:
    scope: "subtree"
    context: "udm:contexts:ou"
target-is-self:
  guardian:builtin:target-is-actor:

permission-sets:  # named permission bundles (can be emulated or actually implemented in
  # FYI: YAML allows references itself, which we can additionally use of permission sets
allow-sensitive-users-user-write:
  - "udm:users/user:read-property-*"
  - "udm:users/user:write-property-*"
  - "udm:users/user:none-property-password"
  - "udm:users/user:none-property-serviceSpecificPassword"
allow-sensitive-groups-group-write:
  - "udm:groups/group:read-property-*"
  - "udm:groups/group:write-property-*"
  # TODO: restrict on sensitive defaults
allow-all-read:
  - "udm:users/user:read-property-*" # my lazyness to not list all permissions here
  - "udm:groups/group:read-property-*" # my lazyness to not list all permissions here
  - ... # further lazyness for the object types
allow-all-write:
  - "udm:users/user:read-property-*"
  - "udm:users/user:write-property-*"
  - "udm:groups/group:read-property-*"
  - "udm:groups/group:write-property-*"
  - ... # further lazyness for the object types

capabilities:  # raw capabilities suitable for re-use/reference, suitable as API for customers
  udm:generic:  # namespace
  users-user-ou-write:
    displayname: "Organizational Unit Admins can administrate all users/user objects in"
    grants-permissions:
      - allow-sensitive-users-user-write
    conditions:
      AND:
        - users-user-objects
        - object-access-underneath-own-ou

  groups-group-ou-write:
    displayname: "Organizational Unit Admins can administrate all groups/group objects in"
    grants-permissions:
      - allow-sensitive-groups-group-write
    conditions:

```

```

AND:
  - groups-group-objects
  - object-access-underneath-own-ou

allow-everything:
  displayname: "Allow everything! Use carefully, only for domain admins!"
  grants-permissions:
    - allow-all-write

allow-self-service-profile-write:
  displayname: ""
  grants-permissions:
    - allow-sensitive-users-user-write
  conditions:
    AND:
      - target-is-self

capability-bundles: # re-useable capability bundles realizing one use case
udm:generic: # namespace
  organizational-unit-administration:
    - udm:generic:users-user-ou-write
    - udm:generic:groups-group-ou-write
  domain-administration:
    - udm:generic:allow-everything
  domain-user:
    - udm:generic:yet-to-be-defined-allow-something
  self-service-profile-administration:
    - udm:generic:allow-self-service-profile-write

role-capability-mapping:
udm:default-roles: # namespace
  domain-users:
    displayname: ""
    capability-bundles:
      - udm:generic:domain-user
  domain-administrator:
    displayname: ""
    capability-bundles:
      - udm:generic:domain-administration
  self-service-profile:
    displayname: ""
    capability-bundles:
      - udm:generic:self-service-profile-administration
  organizational-unit-admin:
    displayname: ""
    capability-bundles:
      - udm:generic:organizational-unit-administration
    capabilities: [] # no further ones
  helpdesk-operator:
    displayname: ""

```

```

    capability-bundles: [] # TODO
    capabilities: [] # TODO
linux-client-join:
    displayname: ""
    capability-bundles: [] # TODO
    capabilities: [] # TODO
    permissions: [] # TODO: idea to allow permissions without a capability (auto-cre

```

Example: Another example is this UDM domain specific language (DSL) in extended BNF grammar inspired by LDAP ACLs. It is easily parseable by a LALR parser, intuitively understandable, human read- and writeable. Most of Guardian concepts are left out here, except for the conditions.

```

named-condition "is-self"
    condition="guardian:builtin:target_is_self"

named-condition "in-global-users-container" # unused example
    condition="udm:conditions:target_position_in" # the guardian condition name
    parameters position="cn=users" scope="subtree" # any key value pair which the guardian

# Organizational Unit Administrators
access by role="udm:default-roles:organizational-unit-admin" context="udm:contexts:positi
    description="Organizational Unit Adminstrators can administrate users and groups in th

to objecttype="container/ou" position="{context}" scope="subtree"
    grant actions="search,read"
    grant properties="*" permission="read"

# mail/domain permission in global mail container
to objecttype="mail/domain" position="cn=domain,cn=mail,{ldap_base}" scope="subtree"
    grant actions="search,read"
    grant properties="*" permission="read"

# user permissions in OU
to objecttype="users/user" position="{context}" scope="subtree"
    description="Write user object in own OU" name="users-user-ou-write"
    grant actions="search,read,create,modify,rename,remove,move"
    grant properties="username,lastname,firstname" permission="write"
    grant properties="password,serviceSpecificPassword" permission="writeonly"
    grant properties="guardianRoles" permission="none"
    grant properties="guardianInheritedRoles" permission="none"
    grant properties="*" permission="write"

# group permissions in OU
to objecttype="groups/group" position="{context}" scope="subtree"
    grant actions="search,read,create,modify,rename,remove,move"
    grant properties="name" permission="write"
    grant properties="password,serviceSpecificPassword" permission="writeonly"
    grant properties="guardianMemberRoles" permission="none"
    grant properties="guardianMemberRoles" values="udm:default-roles:domain-users,udm:de
    grant properties="*" permission="write"

```

```
# Domain Administrators
```

```
access by role="udm:default-roles:domain-administrator"  
  description="Domain Admins are allowed to do anything in the whole domain"  
  to objecttype="*"   
    grant actions="*"   
    grant properties="*" permission="write"
```

```
# Self service use cases
```

```
access by role="udm:default-roles:self-service-profile"  
  to objecttype="*" actions="modify" properties="jpegPhoto,e-mail,phone,roomnumber,departmen
```

Workaround: We are currently going with the second approach and transform this into the required Guardian objects. This also helps us to abstract away the Guardian interface implementation details, so that customers won't see it.

No Caching possible: Capabilities are bound to targets

The API design of Guardian is to either fetch a “access granted (for given target): yes/no” request or a “give me all permission strings granted to the given target”.

It differentiates between “target permissions” and “general permissions”. But in practice most often only target permissions are suitable to be used, otherwise one cannot use any conditions.

If Guardian would be able to give the whole permissions which are possible by an actor, this result would be cacheable. Then we could eliminate a lot of request e.g. when a search is performed twice. And we could use the cache in case Guardian, Keycloak or whatever component is not available - even for a second during some join script run.

The permission system of UMC works that way, that it stores a cache of the granted permissions locally for this case. This makes UMC usable even if no LDAP server is running. And this is an important use case as in case of errors administrators go into the diagnostic module / software update / etc to find out or fix the reasons.

Guardian concepts are too abstract

Guardian doesn't know about basic things a target consists of, which most apps require to implement, e.g. actions like CRUD or more, object types, properties, etc. It just leaves this up to free-form permission strings. Every app in the end can define a own structure. Our company should implement this for all apps in a uniform way.

Therefore the generic Management UI is also not suitable to be used. One must create another UI frontend to realize these concepts so that an Administrator can click on “Is allowed to create/modify” objects of type “teacher” and gets “write” permissions for properties “name, ...”.

Security Impacts

Availability of UDM

Currently, UDM's only dependencies are the OpenLDAP server and UCR DB (local file) availability. Doing authorization via Guardian involves a lot more required components (single points of failure): Keycloak, Guardian Authorization API, Open Policy Agent, PostgreSQL. This opens the way up for:

1. robustness issues

- if one component is not available, and this can happen often during server maintenance and updates, the central UDM identity management is prevented
- do we have enough retry mechanisms to deal with this?

We had a support case which took weeks to solve because Keycloak couldn't be reached.

2. security issues

- Denial of Service has a larger attack surface (attacking just Keycloak makes UDM unusable)
- Denial of Service of UDM affects the other services. Just throw enough requests against various UDM instances to get the other components Co-DoSed.
- Overtaking one component via one vulnerability in the stack allows overtaking the full domain

3. scalability issues

- the above systems are the security foundation of the domain, i.e. they must not run on memberservers or replicas but only on DC Primary and DC backup systems

4. cyclic dependency issues

- e.g. joinscripts
- guardian-management-api → guardian-authorization-api → UDM REST API → guardian-authorization-api

Information disclosure

To prevent information disclosure, the whole implementation must return the same response structure as LDAP would do it. E.g. LDAP says “No Such Object” if a search with a certain search base is done. The `cn=admin` account, with which UDM then runs, has permissions to read all those objects. The UDM interfaces must now raise the same exception and signature. When it would say “Forbidden” instead or the output differs otherwise, an arbitrary user can obtain any information. By combining search bases, search filters and search scopes a lot of attack vectors are possible in every API endpoint and further down, as e.g. syntax classes expose a lot of different possibilities.

This is very tricky to realize, it must be realized everywhere and it must be understood by every further UDM developer so that things don't break when evolving UDM.

The integration of Guardian will make UDM vulnerable to side-channel attacks. If one measures the statistical average time to get such a “no object” response, one can also differentiate whether an object exists or not. In combination with a LDAP filter, this attack can be made very fine granular. Therefore we also have to do guardian requests for objects, which don't exist at all. A http request will add enough time to the response so that this gets a real attack vector.

For large environments doing a search with a base underneath of an object which returns more objects than the configured `sizelimit` will raise an LDAP `SIZELIMIT_EXCEEDED` before any filtering can happen, which will reveal that the object exists.

OPA/Rego doesn't know LDAP DN's

In UDM we use the C library `_ldap` for LDAP DN comparisons to make correct comparisons, as our customer environments use special characters like `+`, `=`, `(`, etc. Depending on the request

and where data origins from we have different DN formats. e.g. `uid = foo \+ bar,cn=users` is equal to `uid=foo \2b bar,cn=user`.

So our checks in OPA need to know these normalization rules when comparing DN's. This is not only from a functional point of view important but also from the security perspective: See the above paragraph; it is required to prevent that actors can enumerate valid LDAP DN's to check if an object exists, they could just give a different DN representation and get different permission results just because LDAP supports both but guardian wouldn't.

We can normalize actor and target DN's before sending them to Guardian, still it must be able to handle special chars. And when it comes to value based comparisons this will break, as we don't have enough Code introspection possibilities in UDM to know if something is a DN or another value. E.g. in OX some values are stored like `$DN || foo || bar`.

Guardian as Policy Information endpoint doesn't add security to the whole system architecture automatically

In some meetings there was the view, that as OPA is a industry standard and proven by large companies, just integrating it will make our architecture more secure. OPA with Rego is a language to easily write a permission system without side effects (clear input and output) and a restricted environment. That's great. But in the end, Guardian is the Policy Information Point and UDM is the Policy Enforcement Point. Therefore the whole permission evaluation is done in the client (UDM), which is more complexity than the whole logic happening in Guardian.

Guardian has no way to trace decisions

A BSI-Grundschutz requirement is that the rule evaluation can be easily traced in logfiles. OPA can log all policy evaluations in detail but the configuration option for `decision_logs` is not integrated in Guardian.

UDM logs all access granting in a structured way.

Guardian debugability

Guardian is the Policy Information Point, UDM is the Policy Enforcement Point so we have two layers where something is decided. With Guardian as an external component involved the debuggability of permission decisions gets hard. We cannot simply trace things in the docker container.

Guardian API design: new/old state of targets not required for filtering

In the Guardian concept the `target` must always be a dictionary with a `new` and a `old` state. While it's not required, to always set a `new` state, this depends on the actual conditions. For some operations like filtering the search results, there is no `old` and `new` state - because the state is equal.

Guardian doesn't clearly document, which builtin conditions operate on which state of the target. The whole `new` and `old` states should just be a client issue, as only they and the conditions give meaning to it.

Performance Impacts

Search results must provide full data to Guardian

We must provide full data, all targets e.g. the search results need to provide all properties and all target roles and inherited roles (depending on the dynamic permissions/conditions). This is not a per-se guardian problem but a general one with the allowed flexibility.

Guardian API design: extensive data format

The Guardian authorization & management API data format is very extensive, e.g. instead of sending : imploded strings, it's always a dictionary {'app_name': ..., 'namespace_name': ..., 'permission_name': }. This requires way more data transfer, serialization and parsing.

Consider, that the whole JSON serialization and de-serialization in 3 components (UDM, Guardian, OPA) will not be negligible.

TODO: evaluate a different JSON library than the one in the standard library.

UDM is synchronous

While the LDAP library supports asynchronicity, our whole UDM code doesn't use it and runs synchronously. This also applies to the guardian rule evaluation, which are HTTP requests. UMC and UDM REST API execute certain blocking operations in a thread, while other things happen on the main thread. These other things are probably now partially blocking the main thread. Daniel proposes to use greenlets. We cannot use the full potential of asynchronicity until UDM isn't designed with async functions.

A local rule evaluation is currently the better approach.

Rule evaluation stays on the client.

The logic to enforce the permissions in UDM requires way more code than the basic functionality Guardian provides. Guardian with OPA is very fast but this doesn't help when the post-processing is done on UDM side. This means that the CPU intensive step stays on client side. Scaling via multiple Guardian instances won't help.

check-permissions vs get-permissions endpoints

Guardian basically only offers two endpoints. To realize our requirements we need them in a combined way. We need to check certain general permissions, some target permissions and receive the whole possible permissions (for reasons see above for wildcard-permissions and afterwards-restrictions). This requires us to do always two requests, which are in the backend doing the same logic but just return different results.

It gets even harder, that we cannot let the permissions checked when we have a search result with a mixed set of objects, for example a search for computer objects will return Domaincontroller Slave and Linux Client objects, with different properties. We cannot specify different permissions we need to check per target. Guardian allows only to check all given permissions for all targets. So we need to make multiple requests.

EDIT: This can (at least partly) be solved by not exposing the module name in a permission, but via a additional check in the condition. This, of course, requires all conditions to be linked

with the **AND** relation. With that, we can send mixed targets to the **get-permission** endpoint and receive specific permissions for each of the target, in one request.

UDM actions do a lot of sub-actions

UDM actions do a lot of sub actions, especially when retrieving objects. We need to check if read permissions for all read references exists so that we don't expose information which is usually not visible to the user.

This requires a lot of checks (via single requests) at different places for one action. UDM is not designed to do a **input representation → just store it in LDAP** operation. It does a lot of sub-operations. UDM Hooks will be even wilder.

Our use case is a secure implementation, not just a “actor can create a user there” and “actor can receive this user”. And this must be specifiable by an Administrator. And we need to tell the administrator, what secure is and what not, with good demo examples.

Management UI Usability problems

The Guardian Management UI is not suitable for the whole UDM domain specific permission/capability assignment. Daniel proposes:

a good UI will display one matrix per UDM module (attribute x permission). Then, the user will not see thousands of permissions/capabilities, but only a few dozen.

So a new UI must be created to be able to work with it. This can by the way, very easily be achieved via a simple UDM module without the need to write a new Javascript frontend.

Managment API bugs and issues

Changing conditions impossible

It's not possible to change a condition, one just get's an Internal Server Error.

Change Request: Fix [univention/dev/projects/authorization-engine/guardian#258](#)

Failed decoding of input JSON data

The Management API crashes permanently randomly due to broken error handling if the Authorization API sends a non successfull response.

Change Request: Fix [univention/dev/projects/authorization-engine/guardian#264](#)

Unreachable API

Sometimes the Apache gateway says the system is not available: **Read timeout:** Nothing occurs in the logfiles.

Change Request: Fix [univention/dev/projects/authorization-engine/guardian#259](#)

Performance of Management API

Our joinscript took more than 35 minutes to create all the default permission strings for all UDM object properties. Each call takes at least 250-1000 milliseconds. Re-running the joinscript took 45 minutes, as Guardian only allows to either create OR modify a permission. UDM provides many permissions for all the UDM modules:

```
apps: 1
capabilities: 152
namespaces: 129
permissions: 11974
roles: 6
```

There is no PUT endpoint, which allows the creation or modification in one idempotent step.

Change Request: All objects in Guardian should support the idempotent PUT endpoint, which creates the object in case it doesn't exist otherwise modify it. Fix [univention/dev/projects/authorization-engine/guardian#265](#)

When creating a permission, which already exists, 100 lines of Traceback are logged.

Change Request: Fix [univention/dev/projects/authorization-engine/guardian#255](#)

Simply storing all permissions on local JSON files was done in nearly 1 second (when not writing debug messages to stdout). A mass-import of the whole structure would help to reduce the performance costs here and would also make sure we always push a idempotent and consistent state for our app "UDM".

Change Request: Guardian should provide a configuration format, like described in example 1 of [No language to describe rules](#), which just allows to push the whole state of an app in a single request.

cyclic dependency problem

The guardian-management-api depends on the guardian-authorization-api, which in turn queries the udm-rest-api, which in turn should query the authorization-api to allow access?

We can implement a solution in UDM which allows certain users (like `cn=admin`) to bypass the Guardian authorization.

Questions

- How should we solve all the guardian problems in a small time frame, where we have a lot of other issues to solve while we also could just create a simple implementation of all this, which doesn't hinder us in the first step and get us going into a compliant solution?
- Do we need for security reasons to evaluate some permissions early in the service, e.g. UDM REST API, UMC-UDM and UDM-CLI? E.g. adding a further additional layer: `udm:udm-rest-api:create-users-user`. Should this be achieved via namespaces or via contexts? Or via extra-data and a condition?
- How to handle situations where one is not allowed to read all the groups of a user (e.g. not the Domain Admins or not the OU2-Teacher group) but he wants to modify the user. The client would send back the received groups and just remove the user from all the other groups not allowed to see.

Conclusion

The core of Guardian is very simple, and re-implemented in 100 lines of Python Code plus another 100 lines for UDM specific conditions.

We don't see the cost-benefit ratio in using the Guardian:

- Why should we use it, when we have to adjust and fix so many things, which requires nearly a Guardian 2.0?
- Why should we use it, when it creates much overhead like sending a lot of strings see-saw?
- If we have to adjust so many things, we have to do in in a generic fashion and have followup work like adjusting manuals, etc.
- Why should we adjust it in a generic way if we don't know the exact use cases so it gets usefull for everyone?
- Are the specific complex things UDM required usefull for every other guardian user or just for UDM? does it give others any value?
- Does it make sense to change the Guardian just to satisfy the UDM needs?

Fixing all these things in Guardian requires more effort than implementing a UDM specific ABAC concept, where rules are stored in LDAP, configurable via certain UDM modules and the UI is automatically rendered by the existing frontend.