univention
be open.

Role and Access Model Strategy

How to continue with Guardian et al.

Univention GmbH

Author: Daniel Tröder

Date: 16.01.2026

univention
be open.

Table of Contents

1 Summary and recommendation

Currently, every Univention software performs and implements authorization independently. For customers, this means they must configure and monitor access to each application separately (→2).

To improve Nubus' manageability and security, we introduce a unified role and access model (RAM) to be used by all Univention software. It allows the customer to interactively and non-interactively define access for all applications and apply that setting to users and groups. It doesn't require the customer to write source code (→3).

We separate the policy engine from the applications and centralize it, because that way it's much better extensible, maintainable, and observable (→3.2).

Not only do we have very diverse use cases: User management (interactive and non-interactive), 3rd party software provisioning, the education domain, Portal, Self-service, SCIM server (→4.1, 6.1). Our use cases also require fine-grained access control. Therefore, we chose to implement an Attribute-based access control (ABAC) system instead of a Role-based access control (RBAC) system. An ABAC system evaluates *attributes* associated with the actor, object, requested operations, and environment, and not just roles, which makes it inherently more expensive. It significantly raises the load on the backend responsible for providing the actors' and targets' data. This is especially true for UDM, and is independent of the authorization engine (→3.3).

A comparison of policy engines and authorization frameworks concludes that the Guardian is currently the only open source system that satisfies our requirements (→5.2.7). The only other system that comes close to what the Guardian offers is Cerbos (→5.2.5). How Cerbos can be used is discussed (→5.5). A more in-depth evaluation is recommended. Regardless of the used policy engine, improvements to the Guardian concept and implementation are necessary (→5.3).

Conclusion: The Guardian is the only viable solution to our needs.

Roadmap (ordered by dependency, later points require earlier ones):

1. Implement conceptual and technical improvements to the Guardian that are independent of the policy engine backend (OPA/Cerbos), including all MUST issues listed in section →5.3 to make the Guardian production ready (160 PT effort). The Guardian can afterwards be used by all components.
2. Create application-specific UIs that use the Guardian Management API.
3. Evaluate Cerbos as a replacement for OPA. This can be done in parallel to 1 and 2.
4. Implement conceptual and technical improvements to the Guardian that depend on the policy engine backend (OPA/Cerbos).

2 The problem

Currently, every Univention software performs authorization independently. Examples:

- The UMC uses policies to find out which modules and endpoints a user or group has access to.
- The UDM REST API's authorization code matches a user's username and groups against a set of UCR variables. (Fine-grained authorization is deferred to OpenLDAP ACLs.)
- The UCS@school library uses the value in a user's extended attribute to determine a user's roles. It authorizes operations on user and group objects. (Fine-grained authorization is deferred to OpenLDAP ACLs.)
- UCS@school applications also use the roles mentioned above. They authorize application operations, such as creating exams, initiating user imports, and managing computer rooms, among other tasks.
- The UCS@school import API requires the user to be a member of groups with a particular set of extended attributes.
- Access to Wi-Fi networks is managed by a script interpreting the value in a user's, group's, or computer's extended attribute.
- The Portal matches a user's groups against the list of groups defined for each portal tile. It defers to the UMC to determine which tiles to display for UMC modules.
- Access to apps is usually authorized by setting a value of an app-specific extended attribute on users or groups.
- OpenLDAP is *not* from Univention, but an integrated third-party software. (Thus, it doesn't belong in this list. It's just here to prevent questions.) It uses LDAP ACLs to configure fine-grained authorization.

Each application/component implements its own authorization.

Not only do they authorize different operations, but they also base their decisions on various data sources (UCR, user attributes, group attributes, and group membership) with differing formats and granularities.

For customers, this means they must configure and monitor access to each application separately.

For example, to allow a user access to the UCS' UMC Computer module, the UCS@school import UI, the Wi-Fi, OpenXchange, and show the respective tiles in the Portal, a customer must configure UMC policies, create groups, edit their attributes and membership, set multiple extended attributes on the user object, and configure Portal objects.

Once the customer has achieved this feat, he can't "export" these settings and apply them to other users, so he won't have to repeat the procedure for each user. Instead, he must use the UCS@school import and create Python

hooks for it, or write UDM hooks, or write Listener modules, or a combination of all that. In any case, he must write source code if he wishes for his users to have consistent access settings.

Then, when he wants to have a second variation of access settings, he can't just copy and edit his "export". He has to edit the source code of his hooks. And those hooks are written in Python, which he may not be familiar with, and they must be installed in the file system with root privileges and deployed on all servers.

Therefore, the current method of defining access is very work-intensive for customers.

3 The solution

Hence, we introduce a unified role and access model (RAM) to be used by all Univention software.

It allows the customer to define fine-grained access rights for all applications and apply that setting to users and groups. It doesn't require the customer to write source code. The settings can be changed using a UI. The settings are not "exported" to a file, but are stored in a database and can be referenced by a name. That "name" is the ID of a role object. A role is a collection of access rights.

External systems can also use the role name (e.g., pass the role of a user managed by an upstream IdP, in an OIDC token to Nubus software).

Univention software can utilize authorization settings to minimize the customer's configuration work. For example, there is no longer a need for customers to configure which groups can view the OX Portal tile or to add UMC policy references to users. Once Univention/the app has defined the required permissions to use the app/module, the Portal can automatically display only the tiles of applications and UMC modules to which a user has access, without the customer having to configure anything.

We choose to implement attribute-based access control (ABAC), instead of a role-based access control (RBAC), for two reasons:

- We must support the concept of *contexts*: A role is only valid in a certain situation, e.g. if the actor or target is inside an OU.
- We have rules that depend on *attributes* of actors and targets, e.g., if a student and a teacher are members of the same class, or app-access depends on a value in an extended attribute.

The data model for rules about directory objects is UDM. LDAP ACLs are not only at the same time too complex and insufficient, they can also not be used to manage access to UDM properties or distinguish between unset and restricted attributes.

3.1 Systems involved

The interpretation of an access right - what you can do with it - is application-specific. Therefore, there is always individual code in each application that determines what to do when a user has a certain access right or not.

But the calculation of the rights a user has, possibly under specific circumstances (conditions), is the same for all applications and should thus be centralized.

The role database (with the collection of access rights as roles) must be centralized. So, with that, we get four systems with dedicated responsibilities:

- Application: Decides what a user can do, depending on what rights it has.
- Authorization engine: Calculates what rights a user has, considering the user, target, operation, and conditions.
- Authorization database: Stores all permissions, roles, conditions, and rules.
- Directory: Stores the roles of all users.

3.2 Why centralize the calculation of a user's rights (the Authorization engine)?

With a centralized authorization component, any part of the application stack can perform permission checks, regardless of the programming language, CPU architecture, operating system, or deployment model.

With modern network stacks, the communication overhead is effectively negligible in all but the most extreme cases.

But let's consider making the code of the Authorization engine into a library and running it in each application.

The most noticeable problems would be extensibility, maintainability, and observability.

Extensibility: Conditions, for example, are source code, and customers or application developers can add them. They could, for example, create a new condition (one that is not part of the library) called "onlyOnWorkdays", to prevent the use of a service on weekends. That condition (its code) must be added to all applications that use the rights-calculation library (the Authorization engine). That means storing the source code on the file systems of all hosts and all containers.

Maintainability: Such a setup is generally undesirable because it significantly increases the complexity of the application. It must manage the policy (rights calculation) engine, including code updates, downloading permission data, and reloading after applying new permission data, as well as expose metrics endpoints and perform other tasks. All of which is outside its application domain. It also creates a dependency on the database format, so that changes to the central authorization system must be coordinated with releases to the embedding application. It's a similar problem to what we have with distributing hooks for UDM in a UCS domain and Nubus for

Kubernetes, just worse, because *every* application (that does authorization) needs it. We'd have the policy engine and all of that policy management and monitoring code running in every component.

Observability: A central authorization system is much easier to monitor than a distributed system. Logging, metrics, and traces can be generated and consumed in one place with consistent formats and known endpoints. Incremental improvements don't require updates to all applications.

Thus, applications should not perform the rights calculation themselves; instead, they should make authorization requests to a central service. This reduces their complexity, enabling centralized management and monitoring of the rights calculation.

Please note that a "central service" can be a distributed system with components and instances running behind a central load-balancer or distributed as application sidecar containers. They still act "as one" and are separate and independent from the application.

3.3 Performance challenges managing directory objects

Customers want to manage access to directory objects using Nubus' public data model: UDM objects.

Thus, we move the authorization layer from the persistence (OpenLDAP) to the backend (UDM). UDM reads objects from the LDAP database and transforms them to UDM objects. After the transformation authorization rules can be evaluated.

This is a significant change over the previous method, where the database did the authorization and returned only objects to UDM the user is allowed to see. Not only didn't UDM have to do the authorization, it also didn't have to transform restricted objects.

So, when managing access policies using the UDM model, *regardless* of the authorization method (ABAC or RBAC) and *regardless* of the policy engine, there is more data to be loaded from the database and more data to be processed by UDM.

In Nubus, this not only increases the load on the database (mostly I/O) but also the time spent on UDM object creation (mostly CPU).

This combination is especially problematic because UDM code is written entirely sequentially (with no parallelism), resulting in increasing latency, *regardless* of scaling.

This problem is *independent* of the authorization method and the authorization engine.

4 ABAC implementation

4.1 Functional requirements / Known use cases

This is a summary of use cases sorted by component. You'll find more details on each of them in appendix “Collected details of known use cases”.

- Management and assignment of roles, users, and groups can be done graphically or via an API.
  - Users inherit roles from their groups.
  - Roles grant access to one, a subset, or all of the following components.
  - The same syntax is used for all "names" (roles, etc.), making it easier for customers and developers to use.
- UDM: Allow fine-grained control of what operations a user can do on UDM objects.
- UMC-UDM: Regular users (not only Domain Admins) can use UMC modules to manage UDM/LDAP objects.
- UDM REST API: See "UDM" above.
- Provisioning: The system can use a user's access rights to determine whether to synchronize data with an application (e.g., OX, Office 365) or not.
- UCS@school: School customers can autonomously create roles that also affect UCS components and apps.
- Apps: Access to third-party apps is managed using RAM roles and permissions.
- Portal: The management of what a Portal shows to users is completely automated.
- Self-Service: Administrators can configure what properties of their account regular users can modify.
- SCIM server: Allow fine-grained control of what operations a user can do on SCIM objects.

4.2 Non-functional requirements

- *Maintainability*: We generally hide technical and database details behind APIs.
  - The Guardian concept defines how roles, capabilities, permissions, contexts, and authorization requests work. That makes it unnecessary for customers to write code.
  - The Guardian Management API must be used to manage RAM objects (such as roles and permissions), hiding details about their technical structure and persistence.

- By using OPA, the Guardian spares Univention the need to invent a policy language, carefully design, implement, test, document, and long-term maintain an evaluation engine (plus tooling, management, etc.).

- *Usability*: The Guardian Management API allows the creation of multiple UIs to manage RAM objects - suitable for different use cases: UDM permissions are fine-grained and may require an advanced UI, the UI to manage UCS@school roles and school-related actions should be in a language appropriate for those customers, and the configuration of who can do what in the Self-Service module can be an uncomplex UI.
- *Availability, Performance, Reliability, Scalability*: ABAC systems are distributed systems. Thus, it is essential to create a performant and resilient setup. All Guardian components scale horizontally and support stateless load balancing.
- *Security*: The Guardian follows OPA's concept of separating the policy engine (making policy decisions) from the application (enforcing those decisions). That simplifies security reviews because the policy engine needs to be reviewed only once, and the policy enforcement code in the applications is minimal and can be standardized.
  UDM is the only one who doesn't profit from this, because until now it could completely rely on OpenLDAP making both the policy decisions and their enforcement, and now how a lot of additional code to take over that responsibility.
- *Observability*: The Guardian APIs allow to centrally monitor all authorization requests, permits and denials. Logs, metrics, and traces are currently insufficient, and should be improved.
- *Development performance*: The separation of the application from the policy engine simplifies application development, thereby increasing development speed.

4.3 Implementation effort in applications

The effort of integrating ABAC into an application is independent of the authorization engine.

The size of the effort depends mainly on the number of resource endpoints that must be protected and the granularity of the rules, as well as the details of the resources and contexts.

4.4 Limiting the complexity

The ABAC policy engine (e.g., Guardian) should not artificially limit the rules, attributes, and environment factors it supports. It should be able to handle many different use cases.

But complexity creates effort. It invites errors, increases documentation, support, tests, and generally raises the maintenance effort.

Thus, we should strive to limit the system's configurability. But, as written above, it can't be done on a general level. It should be done on the individual functional level, use case specific. That's where we know what makes sense and what doesn't. We would express this in UIs and CLIs, where we can "steer" customers.

Examples:

- A UI to manage WiFi access at a school could not only offer to configure different networks for students and teachers, but also the dates and hours during which WiFi can be used. ABAC allows considering environmental factors. And while teachers may work in the school in the evening or during vacations, the WiFi could be deactivated for students during those times. At Univention, we should not offer such a configuration option unless our customers strongly request it. Although it's an artificial and not a technical limit, we should also not offer to take the age (birthday) of WiFi users into account.
- The configuration interface for setting rules for a REST API should offer the option to take a request's HTTP method (GET, POST, …) into account. It should, however, not also offer to use the IP address. The official reasoning would be about security considerations. However, the truth is that more options mean more effort in implementation, testing, documentation, and maintenance.
- For the Self-Service, it should be configurable which properties a user can modify in their own account. We *could* offer to allow different sets of properties for different users, depending on arbitrary rules. Instead, we should offer only one set for all users.

5 Comparison Guardian and alternative implementations

We are seeking a system that enables us to implement all the aforementioned use cases.

5.1 ABAC components

An ABAC system is comprised of the following components that interact in the described way:

1. A user or system requests to take an action on a resource from an application (e.g., searching or modifying a UDM object, starting an exam, or deleting an appointment).
2. The application is the *Policy Enforcement Point* (PEP). It protects access to a resource. The PEP creates a request that contains the user's and the resource's attributes, as well as the operation they wish to execute, and sends it to the PDP. → PEP=Application: UDM REST API, Portal, Open-Xchange, …
3. The *Policy Decision Point* (PDP) evaluates the request with a matching policy and decides whether access should be granted. → PDP: OPA, LocalGuardian

   3.1. The PDP considers data from the PRP and the PIP before issuing access decisions.

   a) The *Policy Retrieval Point* (PRP) stores and serves policies to be used by the PDP.
   → PRP=bundle server run by the Guardian Management API

   b) The *Policy Information Point* (PIP) retrieves the user's and resource's attribute data, environmental conditions, and other contextual information for the PDP.
   → PIP=Guardian Authorization API (uses either data passed in by the request or retrieves data from the UDM REST API)

   3.2. The PDP's answer is sent back to the PEP.
4. The PEP (the application) allows or denies access to the requester (user/system).

The Policy Administration Point (PAP) manages policies and stores them using the PRP.
→ Guardian Management API + PostgreSQL

The Guardian implements all components of an ABAC system (except for the PEP, which is the application itself).

The system is depicted in this diagram:

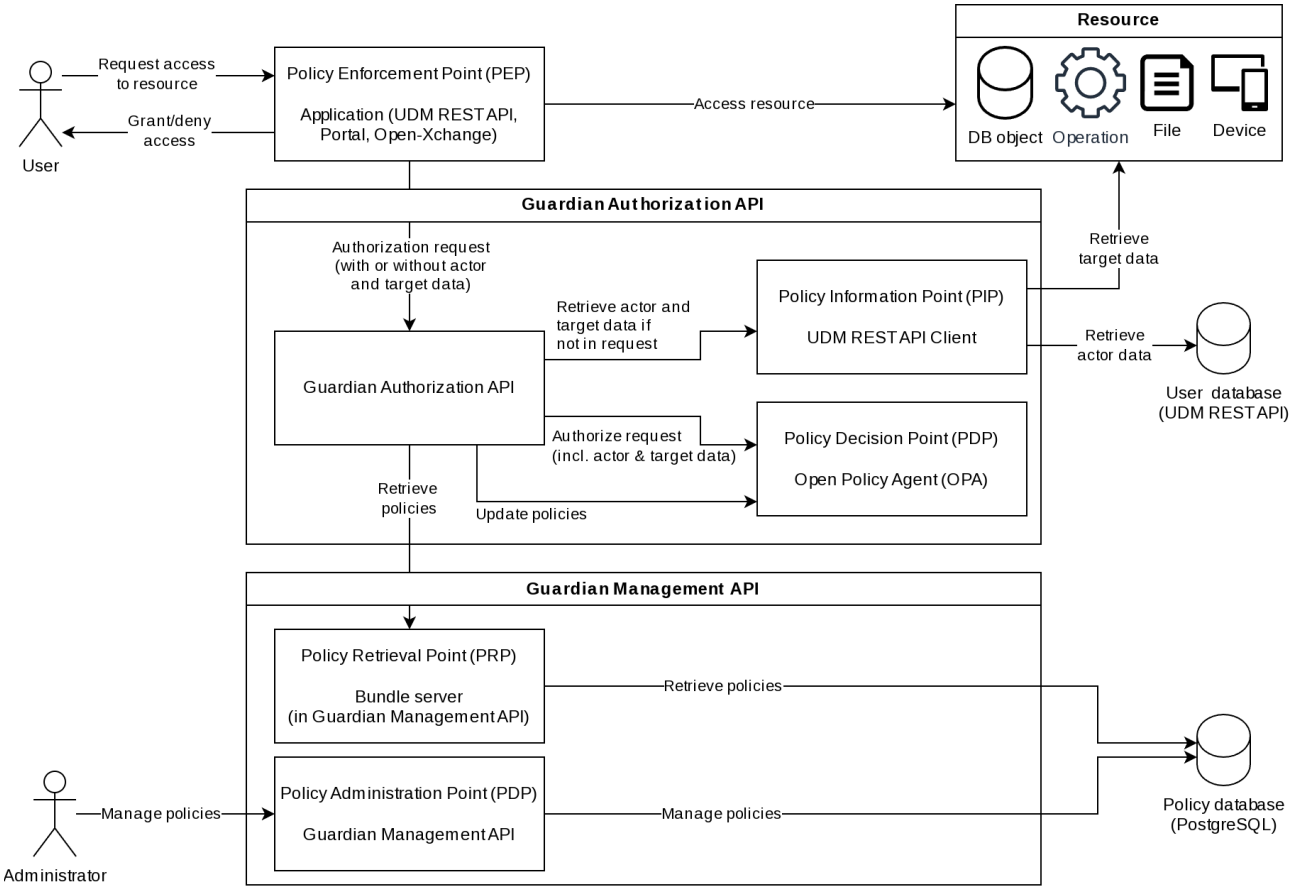

Schaubild 1: The Guardian implements all ABAC components.

5.2 Alternative implementations

A product research on open source policy engines and authorization frameworks yielded the following results.

5.2.1 Guardian

- Administrators only need to define a few Guardian API objects (roles, permissions) to create policies. They can do that using a UI. There is no need to write source code. They can automate it by using a REST API.
- Developers of any application type requiring authorization can use the Guardian.
- Application developers only need to query a REST API to determine if a user has a specific permission or if an operation on a target is allowed. The Guardian even offers to retrieve actor and target data on behalf of the application.
- The Guardian compiles policies from Guardian API objects and distributes them to all policy engine (OPA) instances.
- All Guardian components scale horizontally.

- Extensive documentation exists. But Univention developers report, that in its current state it doesn't help them implement their use cases.

5.2.2 Open Policy Administration Layer (OPAL)

OPAL is an administration layer for OPA. It detects changes to policies and pushes them to OPA (https://docs.opal.ac). It's OSS and free. A commercial offering called "OPAL+" with advanced features exists.

- Administrators who want to define a role, permission, or rule have to write source code in the Rego programming language.
- Developers of any application type requiring authorization can use OPA.
- Application developers must communicate directly with OPA. They have to ship an OPA instance with their application and use OPA's API to query it.
- OPAL compiles policies from Rego code and distributes them to all policy engine (OPA) instances.
- The OPAL server scales horizontally.
- Well documented.

5.2.3 Local Guardian

The "Local Guardian", formerly known as "Guardian-Mock", doesn't fulfill our requirements. Although it is a library that can only be used for non-UDM-related authorization tasks, it is file-based, without a distribution or networking mechanism suited for containers.
It is listed here only for reference.

5.2.4 XACML

The eXtensible Access Control Markup Language (XACML) is an XML-based standard markup language for specifying access control policies. The standard, published by OASIS, defines a declarative, fine-grained, attribute-based access control policy language, an architecture, and a processing model describing how to evaluate access requests according to the rules defined in policies.

Policies written in XACML can be translated to Rego (OPA's policy language), and vice versa.

A handful of OSS PDP engine implementations exist; most are not written in Python. However, there are REST API frontends available for them.:

- ONAP Policy XACML PDP Engine: https://docs.onap.org/projects/onap-policy-parent/en/latest/xacml/xacml.html

- WSO2 API Manager: https://apim.docs.wso2.com/en/latest/manage-apis/design/api-security/authorization/api-authorization/
- AuthzForce (Community Edition): https://authzforce.ow2.org

Regarding the points from before:

- Administrators must write very long XML files following the XACML specification.
- Application developers of any application requiring authorization can use XACML.
- Application developers must write XACML queries. How they communicate with the PDP depends on the product. Most likely, they have to use the REST Profile of XACML.
- Most XACML PDP engine REST frontends I have seen are part of a large ABAC system that includes distributing policies to PDP instances.
- Most large ABAC systems that include an XACML PDP engine scale horizontally.
- Extensive documentation and specifications exists. But the language and system is highly complex.

5.2.5 Cerbos

Cerbos is a language-agnostic, scalable authorization solution that enables writing context-aware access control policies for application resources (https://www.cerbos.dev , https://github.com/cerbos/cerbos).
Policies can be written in either YAML or JSON formats, conforming to their schemas.
Cerbos has extensive documentation and an interactive playground with tutorials:
https://docs.cerbos.dev/cerbos/latest/
It is open source, but also commercially supported. Integrations are available for various programming languages, identity providers, and frameworks: https://www.cerbos.dev/features-benefits-and-use-cases/ecosystem
The PDP, product name "Cerbos PDP", is open source and free of charge.
The management components, product name "Cerbos Hub" (IDE, logging, metrics, CI/CD integration, multi-tenant support), are only available commercially.

- Administrators have to write policies in a declarative style in YAML format. Expressions in conditions are written in the Common Expression Language (CEL). Editor support for the policy schema and the expression language exists. Test are also declarative and written in YAML.
- Developers of any application type requiring authorization can use the Cerbos.
- Application developers only need to query a gRPC or REST API to determine if a user has a specific permission or if an operation on a target is allowed. Cerbos has an endpoint that returns a matching database query (e.g., LDAP filter) to an authorization request.

- Cerbos supports multiple policy distribution mechanisms (disk, Git, database).
- All Cerbos components scale horizontally. Deployment using Helm, Deb, RPM and Docker is supported.
- Extensive documentation exists. Univention developers reported, they could quickly understand its concepts and languages.

→ Ole and Daniel will try Cerbos out in the Hackathon on 04.09.2025 (Issue).

→ We did, and it was a complete success. We migrated the UCS@school user management UI to using Cerbos instead of Guardian in just one day! (Merge Request) The rules are much better to read then with OPA's Rego language, the documentation and the tooling is very good.
There is one important difference: Cerbos doesn't support - by design - the retrieval of all permissions a user has on a target. You always have to send a list of permissions to evaluate. That makes it *incompatible* with the Guardian Authorization API.
It has a very interesting feature: It can generate an abstract database query from an authorization request. When translated into a concrete DB query (e.g., LDAP or SQL), the database would respond with only the objects that have been authorized. This feature has the potential to alleviate many of our performance concerns. It's limited to non-relational queries though.

5.2.6 Casbin

An authorization library that supports access control models like ACL, RBAC, ABAC, ReBAC, BLP, Biba, LBAC, Priority, RESTful for most programming languages, including Python: https://casbin.org/
It is well-documented and can be integrated into our stack.
Unfortunately, its support for ABAC is only rudimentary and, in my opinion, insufficient to fulfill our requirements.

- Extensive and well written documentation exists.

5.2.7 Conclusion

The Guardian is currently the only system that offers customers a UI and API to define roles, permissions, and rules graphically. It compiles them into the language the PDP understands. Such a UI and API would have to be created for the other policy engines as well.

The *only* system that offers the same level of configurability, declarative policies, and support for scaling as the Guardian does, including the required components, is Cerbos.

With Cerberos being the only alternative to the Guardian, which could replace *parts* of its components, not its APIs, today, *the Guardian remains the only viable solution to our needs*.

Not only would any alternative to the Guardian have to be functionally equivalent, but it would also have to be as well integrated into Nubus. Thus, we should continue using the Guardian and develop what's necessary to make it production-ready.

Section →5.5 ”Replacing Guardian or OPA with Cerbos” discusses Cerbos and Guardian/OPA.

5.3 Effort required for making the Guardian production-ready for both UCS and Kubernetes

Redacted. Please see file “Department Development/Roles and Access Management/2025-11-27_RAM_Strategy_v2.odt” in the internal Owncloud.

5.4 UIs to manage roles and permissions

Our use cases are divers. We have different domains (e.g., user, portal, and email administration), different granularity, and different languages or names (e.g., OU vs. school).

While (Microsoft) domain administrators want to work on “users”, “group”, “computers”, and “OUs”, in the educational sector administrators want to manage “schools”, “school classes”, and “pupils”. OpenDesk administrators will expect to find the terms used by OpenProject, Open-Xchange, Nextcloud, and XWiki in our UI, and developers expect HTTP verbs when configuring access to REST APIs.

So, while having a single general UI (like the current Guardian Management UI) is good for getting a complete technical overview, we’ll need application-specific UIs to meet customer demands.

All “RAM UIs” should use the Guardian Management API as backend.

Having a common Management API simplifies frontend development and allows us to switch the policy decision backend (OPA/Cerbos).

5.5 Replacing Guardian or OPA with Cerbos

Replacing the Guardian or parts of it with Cerbos is discussed here with two main goals: reducing system complexity and reducing maintenance effort.

Using Cerbos’ declarative policy language is a nice developer experience. Not only is its documentation easy to understand, but there is also a web-based playground for interactive testing of policies, and tests can be written in a declarative language. The Guardian’s feature to transform its declarative language (the Guardian objects) to Rego isn’t required when using Cerbos *directly*. Thus, the Guardian Authorization and Management APIs are no longer needed. When a policy distribution mechanism is established, the Guardian could be abandoned altogether. This would massively reduce maintenance and documentation efforts.

But that scenario leaves out the customers needs. In section 3 “The solution” we wrote: *[The solution] allows the customer to define fine-grained access rights [..]. It doesn't require the customer to write source code. The settings can be changed using a UI.*

Management API: Although Cerbos’ policies are relatively easy to read and write, they are still source code. And with a system as big and complex as Nubus and UCS@school their number and size will quickly become overwhelming for non-developers. Thus, we still want administrators, application developers, and ProfS to create Guardian objects (using a UI or API), which the Management API translates to the PDP’s language. Cerbos’ policies will be easier to generate than OPA’s Rego code, which will reduce the maintenance burden for developers of the Guardian. But we will not get rid of the Management API component.

Authorization API: What about replacing the Guardian Authorization API with direct calls to Cerbos’ gRPC or REST endpoints? Again, although this has the potential for reducing our code base, we can’t do it. For one, Cerbos doesn’t have an endpoint that fetches actor and target data for clients if they can’t access the UDM REST API themselves. But more importantly, we don’t want to expose technical details of the policy engine to applications. That would bind us to a specific product. The assessment we’re doing right now (which product to use) will only be possible in the future if we have an abstraction layer between the application and the policy engine. Therefore, we won’t eliminate the Authorization API component either.

Policy distribution: Cerbos can read policies from files, Git repositories, and databases. When using a database, it features an Admin API to manage them. In the Guardian system, the Management API produces a bundle server that OPA uses to fetch its policies. The Cerbos Admin API could replace the Guardian’s bundle-server. However, that API lacks granular authorization; access is either all or nothing. The Guardian Management API features fine-grained access controls, as it applies Guardian authorization rules to itself. To implement a networked, secure policy storage, we’ll have to wrap the Cerbos Admin API in a small proxy API that transparently adds authorization. Again, no code or complexity reduction can be achieved.

Switching engines: It’s probably possible to switch transparently from one authorization engine to the other, although the two PDPs have different concepts. OPA, and thus the current Guardian API, has an endpoint to request all permissions an actor has for an operation on a target. Cerbos doesn’t have that. Instead you have to send all permissions you want tested to it. Both queries can be used in most use cases, and one can be translated into the other. Cerbos supports role inheritance with conditions; the Guardian currently doesn’t. Switching from one engine to another may be possible without changing the API or the application. If a transition from OPA to Cerbos is desired, it should be tested if it’s possible to do it transparently – without changing the Authorization and Management APIs and all applications. A gradual migration with different APIs would also be possible because we have a common roles-to-policy compilation component: the Management API.

6 Appendix

6.1 Collected details of known use cases

This list of known use cases should help the reader save a lot of time collecting information from many sources.

6.1.1 Administration of roles, users, and groups

- Administrators can assign roles to users and groups.
- A group's roles are added to the roles stored in each member object.
- A role can grant a user access to any of the components listed below.
- A role can grant access to multiple components.
- Administrators can assign roles to users and groups either graphically or via an API.
- Administrators can manage the access a role grants, graphically or via an API.
- The system defines default roles that can't be changed or deleted.

6.1.2 UDM

Allow fine-grained control of what operations a user can do on UDM objects.

- Applications using UDM are discussed separately. The following is true for Python-UDM and all applications using it.
- The use of delegated authorization, instead of the classic one, can be switched on/off per application using Python-UDM.
- A privileged LDAP account is used instead of the user's account to bypass LDAP ACL evaluation.
    - Audit logs must include requester identity and action metadata, even if the LDAP entry logs only reflect the service account.
- Rules limit the CRUD operations a user can perform on each UDM object type.
- Rules limit which attributes are readable and writable for each UDM object type.
    - Empty attribute values and attributes that the user has no right to read are distinguishable.
- Two default roles, which represent the roles existing without RAM, have been created:

- Domain Administrator (udm:default-roles:domain-administrator): Can perform CRUD operations for every object on every position in the directory.
- Domain User (udm:default-roles:domain-user): Can read his own object.

- For nVZD, three roles have been created:
  - OU Admin (udm:default-roles:organizational-unit-admin): This role enables the delegated management of users, groups, and computer accounts within the assigned Organizational Unit (OU).
  - Computer join account (udm:default-roles:linux-ou-client-manager): Allows the technical staff to onboard Linux clients into the domain by creating machine accounts in a designated OU.
  - Helpdesk (udm:default-roles:helpdesk-operator): Designed for first-level support personnel or service desk agents. This role allows viewing basic user information and resetting passwords within the assigned OU.
- Delegative administration documentation: https://docs.software-univention.de/ext-delegative-administration/5.2/en/
- Issue: Guardian-Based architecture for UDM Result List Filtering: Cross-Team Decision: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/422
- PM Scenario Delegative Administration for State Governments that leverage a data center: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/383
- Epic GI: RAM Integration in nVZD: https://git.knut.univention.de/groups/univention/-/epics/909
- Issue Spike: Evaluate Guardian for delegative administration/product incorporation:
- Documentation in UCS source code: https://git.knut.univention.de/univention/dev/ucs/-/blob/5.2-2/management/univention-directory-manager-modules/README-authorization.md?ref_type=heads

6.1.3 UMC-UDM

Regular users (not only Domain Admins) can use UMC modules to manage UDM/LDAP objects if they have been assigned the appropriate permissions (i.e., the respective roles).

- When searching user or group objects...
  - The rendering of a (paginated) result page must finish in under 1 second (target < 200ms).
  - The UI must support at least 500,000 users in the system without blocking.

- The modules guide users in using predefined filters to enhance their UX, as these can significantly reduce the time required for search operations.
- The user can distinguish an empty attribute value from an attribute that he has no right to read.

6.1.4 UDM REST API

- The delegated administration described above is functional.
- The bulk retrieval (e.g., 150k) of user objects must take less than 100 ms per object.

6.1.5 Provisioning

- The Provisioning system (for synchronizing data to external systems) can use RAM information to determine whether to synchronize data with an application (e.g., OX, Office 365) or not.
- Synchronizing data is an operation involving an actor (the Provisioning Consumer/Listener module) and a target (an internal or external service, such as OX, Office 365, Samba, or Radius). The authorization system can be used for fine-grained control of what is being transmitted (e.g., a customer wants to prevent transferring the user's birthday, mobile phone number, and recovery email address).
- PM Use case ACLs for provisioning APIs: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/302
- PM Use case Nubus Functional Administrator choose objects by guardian roles for SCIM provisioning backend: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/441

6.1.6 UCS@school

This product has implemented RBAC years ago. It suffers from roles having been hard-coded. Adding a role to the whole stack is very expensive (100-150PT) and error-prone. School roles can't be used in UCS automation tooling (UDM hooks, Listener modules).

- School customers wish to create additional roles without having to commission them from Univention.
- The new roles should also be usable for UCS and app-related access control.
- School customers have high data protection requirements. Thus, they require fine-grained access control down to the attribute level.
- Affected Modules/components:
  - Users management UI:

- User management: Users with the appropriate permissions can run operations on users and groups of their schools (similar to UDM's OU admin).
- The role "School Administrator" already exists for that purpose. However, parts of the following operations should now also be executable by users with fewer permissions:
  - CRUD users, groups, computers, computer rooms, printers.
  - (De)activate users.
  - Create reports.
  - Manage Wi-Fi access for users and groups.
- Password reset:
  - School Administrators can reset the passwords of all members of their schools.
  - Teachers can reset the passwords of students in their classes.
- School: Allow users other than Domain Administrators to create a new school and assign a School Administrator.
- Kelvin: Support the above operations in a REST interface, applying authorization as described.
- User imports: Use roles instead of group membership to determine if a user can manage user imports for their schools.
- Exams: Use the new roles instead of the old ones.
- Internet access (HTTP filter): Use the new roles instead of the old ones.
- Material distribution: Use the new roles instead of the old ones.

◦ Rework of Roles & Rights and Web-UI for UCS@school - Requirements: https://filestore.knut.univention.de/owncloud/f/3827821

6.1.7 Apps

Today, users are granted access to apps such as OpenXchange, NextCloud, Radius, or Office 365 by setting a specific value in an extended attribute of their user object. Some apps have finer-grained access control settings or use roles that apply only in certain contexts, e.g. deputy permissions for mailboxes in OX or application roles in WordPress.

- Besides the additional cost of these LDAP attributes, their naming and the format/syntax of the values are often inconsistent.

- The management is also inconsistent, as some settings can be applied to groups, while others must be set individually for each user.
- If an app should be enhanced and requires an additional extended attribute, it implies restarting all LDAP servers in the domain to activate the new LDAP schema. This is undesirable.
- Creating an additional RAM permission, role, or condition, on the other hand, is fast and doesn't incur a service interruption.
- Integration with the Portal is currently very basic. Refer to the "Portal" point below to see how roles improve this.
- Whether access control to apps can be migrated from extended attributes to permissions and roles depends on the integration. If it's a synchronization from Nubus to the app, it's usually possible. If the app actively searches the LDAP, it might not be feasible or difficult.

6.1.8 Portal

The maintenance of what a Portal shows to users can be completely automated.

- Currently, administrators must configure a list of groups on each tile object. The Portal analyzes a user's group membership to determine which tiles to display.
- This leads to customers creating many groups only to manage Portal tiles for their users. Those groups are otherwise useless, but they cause problems because they can become much larger than what OpenLDAP supports with acceptable performance.
    - The openDesk project has a UDM hook that automates the mapping from users' extended attributes to groups, thereby increasing the system's complexity and reducing the performance of UDM.
- What a customer expects is that if a user has been granted access to an app, a Portal tile for the app should be displayed; otherwise, it should be hidden. Although the information exists in the user's LDAP object (or its groups), the Portal can't use it, as it's stored in extended attributes that can't be matched with Portal tiles.
- Using the RAM, full automation can be easily accomplished.
    - The app provider defines a permission that a user must have to access the app. That permission's name is stored in an attribute of the Portal tile the app creates.
    - The Portal queries the authorization engine for the permissions the user has, and displays all tiles whose permissions match.

- The administrator only needs to add the permission to the roles that they wish to have access to the app.

- Issue Portal configuration for admins is massively simplified with a single graphical interface to configure access to applications. -- standardized (Guardian) access rules apply to Portal: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/289

6.1.9 Self-Service

Administrators can configure what properties of their account regular users can modify.

- PM Scenario Self Service content based on Users' roles: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/300

6.1.10 SCIM server

Allow fine-grained control of what operations a user can do on SCIM objects.

- PM Use case Functional Administrator can configure attributes read/writeable in the Self Service: https://git.knut.univention.de/univention/product-management/requirements-management/-/issues/312