VAL

Generated on Thu Oct 24 2024 15:47:40 for VAL by Doxygen 1.9.8

Thu Oct 24 2024 15:47:40

Chapter 1

VAL Overview

Virtuoso provides numerous options for access control - some specific to a particular realm. For instance:

- Protected SPARQL endpoints associated with specific authentication methods, e.g. /sparql-auth, /sparql-oauth, /sparql-webid
- SPARQL roles restrict the types of SPARQL commands a database user may perform.
- Graph level security allows the setting of permissions bit masks to set the read/write (and sponge) permissions on specific RDF graphs to control public or specific user access.

However, OpenLink's preferred generic access control mechanism for Virtuoso is VAL, the Virtuoso Authentication Layer. VAL provides a generic ACL layer which can be used to protect many Virtuoso resources including SPARQL endpoints or graphs, WebDAV resources and access to the Sponger or individual Sponger cartridges.

VAL provides both authentication and access control services to Virtuoso through an internal Virtuoso API or by two public HTTP APIs, a 'standard' HTTP API and a RESTful variant, which manage rules and groups via their URLs. Both are Turtle-based. VAL supports systems like OpenID and a variety of OAuth services such as Facebook, Google, or Twitter.

Typically, users attempting to access a VAL-protected resource must first login and authenticate themselves through a VAL-supplied authentication dialog. After establishing a VAL session, their access permissions on the resource are checked. Virtuoso LDP resources and containers could be protected in this way, once the required VAL hooks are in place.

The supported authentication methods are:

- HTTP authentication (for users with a Virtuoso SQL user login)
- Basic PKI
- `OpenID`
- `WebID + TLS`
- Third party OAuth services (see also Supported 3rd Party Services)

VAL's generic ACL layer is fully RDF-based, storing rules and groups in the triple store in private graphs, and describing rules using `the W3C acl ontology` and the `OpenLink ACL ontology`. The system also supports restrictions which restrict arbitrary values based on the authenticated user. These can be used to limit the number of query results, enforce quotas etc.

In addition to the API VAL will install a new 401_page and 403_page for the /DAV vhost. SSL-enabled sites need to be manually configured via VAL.DBA.create_val_vhosts() to do the same on the https counterpart. The path of the new authentication page is `/val/authenticate`.vsp. This page can of course also be customized. It is recommended to look at its code to tweak the look.

1.1 VAL Low-Level Authentication API

VAL provides the procedures required for any component in Virtuoso to use these authentication methods (see also VAL Authentication API). There are basically two workflows:

1.1.1 VAL OAuth-like Workflow

The OAuth-like workflow applies to OpenID as well as all supported OAuth-based services.

The workflow for a client is always the same irrespective of the service type:

1. The client requests an authentication URL via VAL.DBA.thirdparty_authentication_url()
2. The client navigates to the returned URL allowing the user to authenticate with the 3rd party service.
3. The 3rd party service redirects to the generic callback procedure VAL.DBA.thirdparty_callback() which will complete the authentication procedure by for example requesting an OAuth access token. It will then call a previously configured procedure for custom processing and finally write an HTML 303 redirection result.

For more details of the possible parameters see the procedure documentation.

1.1.2 VAL's Session Support

VAL supports the creation and usage of standard Virtuoso sessions as stored in `VSPX_SESSIONS`. Sessions are created via VAL.DBA.new_user_session() or VAL.DBA.add_sid_to_url().

This is the default mechanism for VAL.DBA.thirdparty_authentication_url(). VAL.DBA.add_sid_to_url() will create a new session id and add it to the given URL.

1.2 VAL High-Level API

In addition to the low-level authentication API VAL also provides an `authentication.vsp` page and two procedures VAL.DBA.get_authentication_details_for_connection() and VAL.DBA.logout() which allow to easily integrate VAL authentication into any Virtuoso application. Create a login link via VAL.DBA.create_login_page_url(), let `authenticate.vsp` do most of the work and check the result via VAL.DBA.get_authentication_details_for↩
_connection().

See Adding VAL Support to a VSP-based Application for details.

1.3 Configuration

Certain aspects of VAL require properly configured SSL endpoints for `/val/api` and `/DAV`. The former provides an https callback URL for OpenID and OAuth (always recommended but also mandatory for Box.com and Salesforce), the latter requires a custom 40x_page to enable all the VAL authentication methods.

VAL provides VAL.DBA.setup_val_host() which allows to create the necessary vhosts for a given host.

Also registry setting `val_always_use_https_callbacks` can be set to `1` for VAL to always use https callback URLs, not only for the services which require it.

See VAL Configuration for more configuration options.

Chapter 2

The VAL ACL Rule and Group System

VAL provides a generic ACL layer which can be used to protect any resource. It is fully RDF-based, storing rules and groups in the triple store in private graphs, and describing rules via `the W3C acl ontology` and the `OpenLink ACL ontology`. The system provides both an internal API (accessible via VSP procedures) and a public HTTP API. In general it is recommended to use the latter as it includes VAL-powered authentication. The following chapters give an overview of the VAL ACL system and its usage in clients.

In addition to rules the system supports restrictions which, instead of granting access to resources, restrict arbitrary values based on the authenticated user. This can be used to limit the number of query results, enforce complex quotas, or any other numeric value. See ACL Restrictions for details.

VAL has special handling for two scopes (Rule Scopes) of resources: graphs in the triple store (scope IRI oplacl:PrivateGraphs, see also VAL.DBA.get_sparql_scope()) and Virtuoso DAV resources (scope oplacl:Dav, see also VAL.DBA.get_dav_scope()). For details see SPARQL ACL Rules - Defining access to private graphs and DAV ACL Rules.

2.1 ACL Rule System Overview

The ACL rule system provides the possibility for any authenticated person (this includes 3rd-party service ids which have no corresponding Virtuoso SQL account) to create ACL rules for any resource they own or that they have been granted the right for. Each rule and each group lives in an application realm. An arbitrary number of application realms can be created or used. The public HTTP API will always use the realm from the authentication information (or fall back to the default `oplacl:DefaultRealm` if the authentication information does not provide an application realm). The internal API has parameters to set the realm in addition to the rule or group details.

The most basic example of an ACL rule below grants read access for `http://www.facebook.com/foobar` to resource `http://me.com/bla:`

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  oplacl:hasAccessMode oplacl:Read ;
  acl:agent <http://www.facebook.com/foobar> ;
  acl:accessTo <http://me.com/bla> ;
  oplacl:hasScope <urn:myscope> .
```

In addition to individual persons ACL rules can grant access for a simple group, a conditional group, or everyone, i.e. make a resource public. The latter is achieved by granting access to `foaf:Agent`:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> acl:agentClass foaf:Agent .
```

Conditional groups do not consist of a list of members but instead define a set of conditions. Every authenticated person matching these conditions is seen as part of the group. See Conditional Groups for details.

2.1.1 Supported Rule Permission Modes

The ACL system is aware of the following permissions - there is, however, no restriction on the permissions to be stored, meaning an application can define their own permission and use that.

- oplacl:Read - The grantee is allowed to read the resource in question.
- oplacl:Write - The grantee is allowed to write or even delete the resource in question.
- oplacl:Sponge - The grantee is allowed to sponge into or from the given graph. This permissions obviously does only apply to SPARQL realm data (see also SPARQL ACL Rules - Defining access to private graphs).
- oplacl:GrantRead - The grantee is allowed to define ACL rules which grant read permissions on the given resource to others.
- oplacl:GrantWrite - The grantee is allowed to define ACL rules which grant write permissions on the given resource to others.
- oplacl:GrantSponge - The grantee is allowed to define ACL rules which grant sponge permissions on the given resource to others. This permissions obviously does only apply to SPARQL realm data (see also SPARQL ACL Rules - Defining access to private graphs).

2.1.2 Individuals - Supported Person Identifiers

IRIs are used to denote (name or "refer to") agents (people, places, software, machines, or other entities capable of mechanized operation) in the ACL system. Any service supported by VAL is also supported in this system. In other words: any agent IRI that can be verified via VAL authentication can be used as the target of a resource access privilge grant. This ranges from Facebook URIs and G+ accounts to Mozilla Persona identifiers and `WebID`s (See the `ODS API documentation` for a full list with examples.).

Internal Virtuoso SQL accounts are denoted by their personal data space URI, even if ODS is not installed. This means that SQL user `foobar` is denoted by personal URI `http://HOST/dataspace/person/foobar#this`.

2.1.3 Groups

Simple groups which consist of a set of members are defined as `foaf:Group` (or more presise its sub-class oplacl:StaticGroup which is merely used for easy distinction from conditional groups - oplacl:ConditionalGroup).

Statis groups are defined as follows:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .

<#group> a foaf:Group, oplacl:StaticGroup ;
  foaf:name "My Friends" ;
  foaf:member <http://www.linkedin.com/in/bugsbunny> ,
              <acct:123456789@dropbox.com> .
```

2.1.3.1 Conditional Groups

A conditional group does not consist of a set of members but a set of conditions. Each personal IRI matching all of these conditions is seen as being in that group. As such, the member of a conditional group are not a fixed set and can change at any point in time.

Conditional Groups have their roots in WebID authentication which means that many of the possible conditions refer to details of an X.509 client certificate. But the very generic SPARQL ASK query condition type allows to create virtually any condition type possible.

The following examples give an idea of the possibilities of conditional groups. The full set of possible conditions can be taken from the OpenLink ACL Ontology.

Example: Query condition

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .

<#group> a oplacl:ConditionalGroup ;
  foaf:name "Everyone I know" ;
  oplacl:hasQuery """
    ask where {
      graph <urn:mygraph> {
        <http://www.facebook.com/me> foaf:knows ^{uri}^ .
      }
    }
  """ .
```

This condition consists of a simple sparql ask query which checks if the authenticated user is known to a certain other identity as claimed in a certain graph. If `urn:mygraph` was a private graph controlled by the owner of the rule, then the group would be defined by the triples in that graph. The special notation `^{uri}^` is a placeholder for the authenticated personal IRI. Another supported placeholder is `^{graph}^` which refers to the graph containing the WebID profile data.

2.1.4 Recursive Rules

VAL supports recursive rules using oplacl:RecursiveAuthorizarion based on two concepts: file-system type URI-based recursion and relation-based recursion.

2.1.4.1 Recursion Based On The URI

A recursive rule grants access to all sub-resources which start with the same prefix. This is typical for file systems: a recursive rule that grants access to `http://HOST/DAV/home/foobar/test/` will also grant access to all children like `http://HOST/DAV/home/foobar/test/hello.txt`.

2.1.4.2 Recursion Based On Relations

dcterms:hasPart relations in the VAL ACL schema graph `urn:virtuoso:val:acl:schema` (VAL.DBA.get↩
_acl_schema_graph()) are taken into account. That means that a rule granting access to something like `urn:test` in combination with a triple like

```
sparql
prefix dcterms:   <http://purl.org/dc/terms/>
insert into <urn:virtuoso:val:acl:schema> {
  <urn:test> dcterms:hasPart <urn:test:child> .
  <urn:test:child> dcterms:hasPart <urn:test:grandchild> .
}
```

will grant access to `urn:test:child` and `urn:test:grandchild` also.

2.1.5 Resource Ownership

As mentioned above an authenticated user can only create ACL rules for resources they own or for which the owner granted them the corresponding grant permission (see Supported Rule Permission Modes for details). Resource ownership is stored in special graphs. These graphs are controlled by the application. Typically they are private graphs which need to be registered with the system via VAL.DBA.add_ownership_graph() and removed via VAL.DBA.remove_ownership_graph(). The resource ownership in these graphs is controlled by the application (not by VAL - the only exception is SPARQL, i.e. ownership of named graphs. see also SPARQL ACL Rules - Defining access to private graphs) via triples like

```
<https://plus.google.com/1056872387> foaf:made <http://HOST/something> .
```

This triple states that the Google Plus account `https://plus.google.com/1056872387` owns resource `http://HOST/something`.

Warning

> Be aware that graph ownership for named graphs is controlled by VAL for internal reasons. Compare VAL.↩ DBA.add_graph_ownership(), VAL.DBA.set_graph_ownership(), and VAL.DBA.remove_graph_ownership().
>
> DAV resource ownership is the second special case in VAL. Instead of checking the ownership graphs VAL will check the actual `SQL` ownership of the resource and compare that to the authenticated user. Non SQL-users (3rd-party accounts) require `oplacl:GrantRead` etc. permissions instead. VAL uses a hook into the Virtuoso DAV system which will create grant ACL rules should a 3rd-party account create a new resource.

Tip: Since `dba` is allowed to create rules for any resource without restrictions one could even do without ownership by granting `oplacl:GrantRead` and friends to the "owner" or a resource. Even more important this allows for situations in which a user owns the data in a graph but is not allowed to change it via SPARQL - as is the case with ODS graphs which are controlled and populated through the ODS SQL tables.

2.1.6 Application Realms

Each ACL rule, group, and restriction is defined in a specific application realm which is stored with the rule or group using the oplacl:hasRealm and with each restriction using the oplres:hasRealm property. Each application realm defines a distinct set of rules, groups, and restrictions. In the case of the public HTTP API the realm is taken from the authentication information, meaning that all API functions will only handle rules, groups and restrictions from the current realm. The internal API allows to specifiy the realm as a parameter which makes it more powerful (but with great power comes great responsibiliy - do not use it lightly).

The default realm is `oplacl:DefaultRealm`. This will use be used by VAL if no realm is specified.

2.1.6.1 Virtual Host Specific Realm

The application realm can be configured in the virtual host options using the keyword `app_realm`. This will make VAL default to the configured realm rather than `oplacl:DefaultRealm`.

If one for example wanted to create a SPARQL endpoint which used a different set of ACL rules and groups from the default `/sparql` endpoint, one could create the corresponding virtual dir as follows:

```
DB.DBA.VHOST_DEFINE (
  lpath=>'/mysparql',
  ppath=>'/DAV/VAD/val/sparql',
  is_dav=>1,
  def_page=>'sparql.vsp',
  vsp_user=>'dba',
  ses_vars=>0,
  opts=>vector (
    '401_page', '401.vsp',
    '403_page', '403.vsp',
    'app_realm', 'urn:virtuoso:val:realms:myrealm'
  ),
  is_default_host=>0
);
```

VAL.DBA.get_authentication_details_for_connection() will use the configured realm if no other is provided by the application page.

Warning

> Be aware that for this to work properly the authentication page needs to be served from the same virtual host. Otherwise it will simply use the default realm again. See Adding Login and Logout Links for an example.

2.1.7 Rule Scopes

Each ACL rule has a scope like `oplacl:hasScope oplacl:PrivateGraphs`. This scope groups ACL rules by the type of resource they protect.

Scopes can be any arbitrary URI the application chooses to use. Ideally, however, they should be defined in the private acl schema graph `urn:virtuoso:val:acl:schema` (VAL.DBA.get_acl_schema_graph()). A typical scope definition looks as follows:

```
oplacl:Query a oplacl:Scope ;
  rdfs:label "Query ACL Scope" ;
  rdfs:comment """Query ACL scope which contains all ACL rules granting permission to perform SQL or SPARQL
      operations
    in general. The latter is complemented by the private named graphs scope which contains rules for named
      graph access.""" ;
  oplacl:hasApplicableAccess oplacl:Read, oplacl:Write, oplacl:Sponge, oplacl:CreatePublicGraph,
      oplacl:CreatePrivateGraph ;
  oplacl:hasDefaultAccess oplacl:Read, oplacl:Write, oplacl:Sponge .
```

All scopes stored in the VAL scope graph can be accessed via convinience procedures VAL.DBA.acls_enabled↩
_for_scope() and VAL.DBA.get_default_access_for_scope() which allow applications to support disabling of ACL checks.

VAL itself defines three scopes by default for which it contains optimizations:

- `private named graphs` (scope IRI oplacl:PrivateGraphs, see also VAL.DBA.get_sparql_scope()) groups rules protecting named graphs. This scope needs to be used for ACL rules if one wants to use VAL's SPARQL callback to enforce ACL rules. See SPARQL ACL Rules - Defining access to private graphs for details.
- `query` (scope IRI oplacl:Query, see also VAL.DBA.get_query_scope()) groups ACL rules to grant general permission to perform SQL or SPARQL queries, including access to the Virtuoso Sponging engine.
- `dav` (scope IRI oplacl:Dav, see also VAL.DBA.get_dav_scope()) groups rules protecting dav resources, i.e. files and folders.

Additional scopes include:

- The Sponger Cartridges scope (oplacl:SpongerCartridges) groups ACL rules to grant permissions to use specific Sponger Cartridges. By default (i.e. when the scope is disabled) everybody is allowed to use all cartridges.
- The OAuth Scope (oplacl:OAuth) groups ACL rules granting access to the OAuth server part of VAL. This includes the creation of OAuth applications. By default anyone can write which means create OAuth applications.

Scopes can easily be enabled and disabled in a certain realm by setting the corresponding value in the scope graph. If one for example wanted to enable system-wide evaluation of SPARQL rules in the default realm:

```
sparql
prefix oplacl: <http://www.openlinksw.com/ontology/acl#>
with <urn:virtuoso:val:config>
delete {
  oplacl:DefaultRealm oplacl:hasDisabledAclScope oplacl:PrivateGraphs .
}
insert {
  oplacl:DefaultRealm oplacl:hasEnabledAclScope oplacl:PrivateGraphs .
};
```

Be aware though that VAL's ACL rule checking procedures typically do not evaluate these scope settings. It is up to applications to enforce them. The only exception are the Virtuoso ACL hook procedures and the `/sparql` implementation which can be seen as clients to VAL themselves.

2.1.8 Storage of Rules and Groups

Rules and groups are stored in private named graphs within the triple store. Each application realm typically has its own set of graphs for rules, groups, and restrictions. These graph IRIs can be customized as explained in Customizing the ACL Graphs.

The default graphs use the default hostname (`HOST` in the example below) of the Virtuoso instance and build a URI based on that and the realm URI.

Example: The default graph which stores the rules in the default realm is the following:
```
http://HOST/acl/graph/rules/http%3A%2F%2Fwww.openlinksw.com%2Fontology%2Facl%23DefaultRealm
```

Ideally one does not have to care about the graphs or how rules are stored, unless one wants to manage them without the API for whatever reason (not recommended since it circumvents the ownership checks).

Should one choose to manually manage rules it is important to note that the API adds the used realm to all created entities. For rules and groups:
```
<#rule> oplacl:hasRealm oplacl:DefaultRealm .
```

and for restrictions:
```
<#rest> oplres:hasRealm oplacl:DefaultRealm .
```

This essentially means that one could theoreticall use a single graph for all realms and all types of ACL resources.

2.2 Querying Rules/Checking Permissions

The ACL API provides several ways to check for permissions. For internal use there are VAL.DBA.check_acls↩
_for_resource() and friends which return a mapping of resources to their modes, as well as VAL.DBA.find_acl_↩
permissions_basic() and friends which create a result set instead that can be looped over.

Alternatively HTTP clients can use VAL.VAL.acl.permissions.list() to get the permissions for a specific resource for the authenticated user.

As always using the HTTP API is recommended but vsp application developers might find it simpler to use the internal API.

Permissions for named graphs are a special case which is described in SPARQL ACL Rules - Defining access to private graphs and Enforcing SPARQL ACL Rules.

2.3 SPARQL ACL Rules - Defining access to private graphs

The ACL system is a generic system which allows to define rules for any type of resource. To make life easier for developers VAL does handle sparql ACL rules as a special case.

As such VAL defines one special scope oplacl:PrivateGraphs (VAL.DBA.get_sparql_scope()) which is used to store rules and groups pertaining named graphs. In addition ownership of named graphs is handled as a special case directly by VAL. Compare VAL.DBA.add_graph_ownership(), VAL.DBA.set_graph_ownership(), and VAL.DBA.↩
remove_graph_ownership().

In order to make use of these ACL rules the `sql:gs-app-callback` SPARQL pragma needs to be used as follows:
```
define sql:gs-app-callback "VAL_SPARQL_PERMS"
```

This will tell the query engine to use the VAL graph security callback for graph permission checks. In addition the authenticated service id needs to be set with the `sql:gs-app-uid` pragma:

```
define sql:gs-app-uid "http://www.facebook.com/foobar"
```

This is the id which is used to evaluate ACL rules.

The special `uid` value `"nobody"` is supported to represent the fact that no user has logged in. In that case only public rules will be applied, ie. only graphs that are public according to the acl rules are accessed:

```
define sql:gs-app-uid "nobody"
```

If system permissions for `SQL` users should be taken into account (in addition to the ACL rules) - obviously this only applies to service ids that actually map to a `SQL` account - then the mapped username (or uid) needs to be set via:

```
connection_set ('val_sparql_uname', 'foobar');
```

Typically one would use the value provided by VAL.DBA.get_authentication_details_for_connection().

Also the application realm for the ACL rules can be changed from the default `oplacl:DefaultRealm` via a connection setting before issuing the query:

```
connection_set ('val_sparql_rule_realm', 'urn:myrealm');
```

Finally it is highly recommended to cache the `WebID` profile (in the case of WebID authentication) in a temporary graph and set that graph via:

```
connection_set ('val_sparql_webid_graph', tmpWebidGraph);
```

That way the callback can pick it up and reuse it for all permission checks. Otherwise the profile has to be fetched for every permission check which results in a considerable performance drop. The simplest way to achieve this is to let VAL.DBA.get_authentication_details_for_connection() fetch the data by providing the temporary graph IRI to it and clearing the graph after executing the sparql query.

Warning

> Be aware that ACL rules enforced via the callback only apply to private graphs. In other words for any graph Virtuoso SQL user `nobody` has access to ACLs will not be evaluated.

See Enforcing SPARQL ACL Rules for an example.

It is recommended to run sparql queries as user `dba` since the Virtuoso graph security system only allows to lower permissions via the callback, not raise them. Thus, we need a user that has access to all graphs, including private graphs.

2.4 DAV ACL Rules

The ACL system is generic in that it can be used to protect any resource clients with to protect. It does, however, include some special handling of Virtuoso DAV resources:

- VAL supports two types of DAV URLs to declare rules:
  1. The internal URLs using the absolute path in the DAV system grant generic access. An example looks like `dav:/DAV/home/demo/foobar.txt`.
  2. The URLs as seen by http clients which only grant access via that particular protocol (`http` vs. `https`) and virtual dir. An example could look like `http://www.host.com/files/demo/foobar.↩ txt` which would use a virtual dir `/files` -> `/DAV/home`.
- Resource ownership for DAV resources (when using the correct DAV scope VAL.DBA.get_dav_scope()) is checked via the unix-style permissions in the DAV system itself rather than checking the ownership graphs.

By installing VAL three hook procedures are created which integrate the ACL system with DAV:

- DB.DBA.DBEV_CHECK_CONNECTION_AUTHENTICATION()
- DB.DBA.DBEV_CHECK_PERMISSIONS()
- DB.DBA.DBEV_RES_CREATION_POST()

These procedures are used by the DAV system to check the ACLs setup in the DAV scope (VAL.DBA.get_dav_↩ scope()). The most interesting hook is DB.DBA.DBEV_RES_CREATION_POST(). It will create the necessary grant rules which allows a 3rd-party account without a connected SQL account to control newly created resources. This is very important as account like that cannot own DAV resources in a classical way. In fact, new resources created in such a context will have classical unix-style permissions of `000000000-` and be owned by the `nobody` user. This ensures that only the creating user can actually read and write those resources.

See ACL Rule Examples for examples of DAV access rules.

2.5 ACL Restrictions

Besides classical ACL rules VAL allows to create restrictions. Restrictions can be seen as configuration values that are specific for a given person. Other than rules they do not grant rights but maximum or minimum limits on certain values. A typical example are request rates on an HTTP connection.

Restrictions, like rules, have a resource which they restrict. This is denotes by `oplres:hasRestricted↩ Resource`. In addition one resource can have multiple restrictions attached to it by using the optional `oplres↩ :hasRestrictedParameter`.

The internal VAL API provides some procedures to check restriction values:

- VAL.DBA.find_restrictions_min()
- VAL.DBA.find_restrictions_max()
- VAL.DBA.find_restrictions()

In addition there is the public HTTP and RESTful APIs.

2.6 The Public ACL HTTP API

See The RESTful ACL API and VAL Public ACL HTTP API.

2.7 The Internal ACL API

See VAL Internal ACL API.

2.8 The Internal ACL Utility API

See VAL Internal ACL Utility API.

2.9 ACL Examples

2.9.1 ACL Rule Examples

Example: Simple ACL rule sharing named graph urn:foobar with facebook ID http://www.facebook.com/john.tester:

The scope oplacl:PrivateGraphs is reserved by VAL to handle rules granting permissions to named graphs.

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  oplacl:hasAccessMode oplacl:Read ;
  acl:accessTo <urn:foobar> ;
  acl:agent <http://www.facebook.com/john.tester> ;
  oplacl:hasScope oplacl:PrivateGraphs .
```

Example: ACL rule allowing Mozilla Persona identity harry@gmail.com to create ACL rules on named graph urn:foobar:

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  rdfs:label "Allowed to grant read for urn:foobar" ;
  oplacl:hasAccessMode oplacl:GrantRead ;
  acl:accessTo <urn:foobar> ;
  acl:agent <acct.persona:harry@gmail.com> ;
  oplacl:hasScope oplacl:PrivateGraphs .
```

Example: ACL rule granting read/write permissions to a group of ids:

Given that a group with IRI http://HOST/acl/groups/42 (substitute HOST with the appropriate value) was created before.

For examples of groups including conditional ones see ACL Group Examples.

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  oplacl:hasAccessMode oplacl:Read, oplacl:Write ;
  acl:accessTo <urn:foobar> ;
  acl:agent <http://HOST/acl/groups/42> ;
  oplacl:hasScope oplacl:PrivateGraphs .
```

Example: ACL rule which grants generic access to DAV resource /DAV/home/demo/foobar.txt for LinkedIn ID horstmeier:

This rule allows horstmeier to access the DAV resource via any protocol and any virtual directory.

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  oplacl:hasAccessMode oplacl:Read ;
  acl:accessTo <dav:/DAV/home/demo/foobar.txt> ;
  acl:agent <http://www.linkedin.com/in/horstmeier> ;
  oplacl:hasScope oplacl:Dav .
```

Example: ACL rule which grants http-only access to DAV resource /DAV/home/demo/foobar.txt for LinkedIn ID hildemeier:

This rule allows hildemeier to access the DAV resource only via http.

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  oplacl:hasAccessMode oplacl:Read ;
  acl:accessTo <http://HOST/DAV/home/demo/foobar.txt> ;
  acl:agent <http://www.linkedin.com/in/hildemeier> ;
  oplacl:hasScope oplacl:Dav .
```

Example: ACL rule which grants http-only access to DAV resource `/home/demo/foobar`.txt for SQL user `joey`:

This rule allows `joey` to access the DAV resource only via http and only via the `/home` virtual dir serving that path.
```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .

<#rule> a acl:Authorization ;
  oplacl:hasAccessMode oplacl:Read ;
  acl:accessTo <http://HOST/home/demo/foobar.txt> ;
  acl:agent <http://HOST/dataspace/person/joey#this> ;
  oplacl:hasScope oplacl:Dav .
```

Example: Recursive ACL rule which grants access to all DAV resources below `/DAV/home/demo/Public/` to everyone (Caution: recursive DAV rules should always use a trailing slash for the resource URL):
```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix acl: <http://www.w3.org/ns/auth/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#rule> a acl:Authorization, oplacl:RecursiveAuthorizarion ;
  oplacl:hasAccessMode oplacl:Read ;
  acl:accessTo <http://HOST/home/demo/Public/> ;
  acl:agentClass foaf:Agent ;
  oplacl:hasScope oplacl:Dav .
```

Warning

Should you decide to create rules manually using `SPARQL INSERT` (not recommended because it circumvents the security checks for ownership) be aware that the realm needs to be set on each rule:
```
<#rule> oplacl:hasRealm oplacl:DefaultRealm .
```

2.9.2 ACL Group Examples

Example: A simple group which enumerates its members
```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a foaf:Group ;
  foaf:name "Some people" ;
  foaf:member <http://dduck.wordpress.com> ,
              <http://peterparker.tumblr.com/> .
```

Example: A conditional group which includes every validated X.509 client certificate. In other words: A rule which uses this group as an agent grants access to anyone who can provide a verifiable X.509 client certificate.
```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a oplacl:ConditionalGroup ;
  foaf:name "Valid Client Certificates" ;
  oplacl:hasCondition [
    a oplacl:GroupCondition, oplacl:GenericCondition ;
    oplacl:hasCriteria oplacl:CertVerified ;
    oplacl:hasComparator oplacl:EqualTo ;
    oplacl:hasValue 1
  ] .
```

Example: A conditional group which includes every validated `WebID`. In other words: A rule which uses this group as an agent grants access to anyone who can provide a valid WebID certificate.
```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a oplacl:ConditionalGroup ;
  foaf:name "Valid WebIDs" ;
  oplacl:hasCondition [
    a oplacl:GroupCondition, oplacl:GenericCondition ;
    oplacl:hasCriteria oplacl:WebIDVerified ;
    oplacl:hasComparator oplacl:EqualTo ;
    oplacl:hasValue 1
  ] .
```

Example: A conditional group which includes every validated identity (aka NetID). In other words: A rule which uses this group as an agent grants access to anyone who can provide a valid identifier (NetID). This includes `WebID`s, Facebook accounts, OpenIDs, etc..

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a oplacl:ConditionalGroup ;
  foaf:name "Valid Identifiers" ;
  oplacl:hasCondition [
    a oplacl:GroupCondition, oplacl:GenericCondition ;
    oplacl:hasCriteria oplacl:NetID ;
    oplacl:hasComparator oplacl:IsNotNull ;
    oplacl:hasValue 1
  ] .
```

Example: A conditional group which includes identities based on a `SPARQL ASK` query. The conditional group defines one condition that has a query which simply checks if the authenticated user is mentioned as an object in a `foaf:knows` statement. So any identifier which is `foaf:known` to ODS user `trueg` will be part of the group.

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a oplacl:ConditionalGroup ;
  foaf:name "People I foaf:know" ;
  oplacl:hasCondition [
    a oplacl:GroupCondition, oplacl:QueryCondition ;
    oplacl:hasQuery "ask where { <http://trueg.dyndns.org:8890/dataspace/person/trueg#this> foaf:knows
      ^{uri}^ }"
  ] .
```

Example: A conditional group which includes any `WebID` that claims to be a person in their profile. `^{graph}^` will be replaced with the profile graph corresponding to the authenticated identity, while `^{uri}^` will be replaced with the authenticated service identifier URI.

```
@prefix oplacl: <http://www.openlinksw.com/ontology/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a oplacl:ConditionalGroup ;
  foaf:name "Persons" ;
  oplacl:hasCondition [
    a oplacl:GroupCondition, oplacl:QueryCondition ;
    oplacl:hasQuery "ask where { graph ^{graph}^ { ^{uri}^ a foaf:Person } . }"
  ] .
```

Warning

> Should you decide to create groups manually using `SPARQL INSERT` (not recommended because it circumvents the security checks for ownership) be aware that the realm needs to be set on each group:
> `<#group> oplacl:hasRealm oplacl:DefaultRealm .`

2.9.3 ACL Restrictions Examples

Restrictions, like rules, always apply to an agent or agent class.

Example: Restriction which sets a maximum of 1000 result set rows for the Virtuoso /sparql endpoint:

```
@prefix oplres: <http://www.openlinksw.com/ontology/restrictions#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#rest> a oplres:Restriction ;
  rdfs:label "Restrict max result set rows" ;
  oplres:hasRestrictedResource <urn:virtuoso:sparql:maxresultsetrows> ;
  oplres:hasAgentClass foaf:Agent ;
  oplres:hasMaxValue "1000"^^xsd:decimal .
```

Example: Restriction which makes an exception to the above rule by allowing a group of premium customers to query more rows:

```
@prefix oplres: <http://www.openlinksw.com/ontology/restrictions#> .

<#rest> a oplres:Restriction ;
  rdfs:label "Restrict max result set rows for premium customers" ;
  oplres:hasRestrictedResource <urn:virtuoso:sparql:maxresultsetrows> ;
  oplres:hasAgent <http://HOST/acl/groups/9> ;
  oplres:hasMaxValue "100000"^^xsd:decimal .
```

This example depends on a static group which enumerates the premium customers:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#group> a foaf:Group ;
  foaf:name "Premium customers" ;
  foaf:member <http://dduck.wordpress.com> ,
              <http://peterparker.tumblr.com/> .
```

Warning

> Should you decide to create restrictions manually using `SPARQL INSERT` (not recommended because it circumvents the security checks for ownership) be aware that the realm needs to be set on each restriction:
> `<#rest> oplres:hasRealm oplacl:DefaultRealm .`

2.9.4 ACL API Examples Using Curl

The following examples assume that a user session has been created via one of the login pages like `/sparql` or, if using a non-browser client, via `/val/api/login` (see VAL.VAL.login) or through ODS. Other authentication mechanisms like HTTP digest login or OAuth are also supported, but they will always result in usage of the default realm.

Example: Obtain a session ID using /val/api/login with WebID authentication:

```
# curl -ikL --cert-type P12 --cert mycert.p12:mypassword https://HOST/val/api/login?service=webid

HTTP/1.1 200 OK
...
Set-Cookie: sid=c295ed6ff661...; path=/; expires=Mon, 13 Nov 2017 14:34:18 GMT
Content-Length: 181

{"status": "success", "httpcode": "200"  "message": "Login successful. Logged in as:
      http://WEBID_HOST/sample_webid.ttl#identity" }
```

Example: Obtain a session ID using /val/api/login with HTTP digest authentication:

```
# curl -iL -u dba:dba --digest http://HOST/val/api/login?service=digest

HTTP/1.1 401 Unauthorized
...
WWW-Authenticate: Digest realm="http://www.openlinksw.com/ontology/acl#DefaultRealm",
      domain="/val/api/login", nonce="dc4268...", opaque="5ebe22...", stale="false", qop="auth",
      algorithm="MD5"
Content-Length: 0

HTTP/1.1 200 OK
...
Set-Cookie: sid=28b9c4af7e4...; path=/; expires=Thu, 16 Nov 2017 10:20:27 GMT
Content-Length: 92

{ "status": "success", "httpcode": "200"  "message": "Login successful. Logged in as dba." }
```

Example: Create an ACL rule using the `GET`-based ACL API:

```
# curl "http://HOST/val/api/acl.rule.new?sid=SID \

      &ruleData=%3Ax+a+acl%3AAuthorization+%3B%0A++foaf%3Aname+%22Allowed+to+grant+read+for+urn%3Afoobar%22+%3B%0A++\

      acl%3Amode+oplacl%3AGrantRead+%3B%0A++acl%3AaccessTo+%3Curn%3Afoobar%3E+%3B%0A++acl%3Aagent+%3Cacct.persona%3Aharry%40g
  %3E+%3B%0A++oplacl%3AhasScope+%3Curn%3Avirtuoso%3Aval%3Ascopes%3Asparql%3E+.\
  &format=text%2Fturtle"
```

Example: Create an ACL rule using the `REST`ful API. reading the rule from file `rule.ttl`:

```
# curl -X POST -H "Content-Type: text/turtle" -d @rule.ttl http://HOST/acl/rules?sid=SID
```

Example: Add a member to a group using the `REST`ful API:

```
# curl -X PATCH -H "Content-Type: text/turtle" -d "<#group> a foaf:Group ; foaf:member
      <acct:123456@dropbox.com> ." http://HOST/acl/groups/42?sid=SID
```

Example: Set the member of a group, overwriting the existing ones:

```
# curl -X PUT -H "Content-Type: text/turtle" -d "<#group> a foaf:Group ; foaf:member
      <acct:123456@dropbox.com> ." http://HOST/acl/groups/42?sid=SID
```

Example: List the groups in the authentication realm scoped to the `sid:`

```
# curl http://HOST/acl/groups?sid=SID
```

2.9.5 ACL API Examples Using jQuery

Example: Create an ACL rule using the RESTful API:

```
var ruleData = '<#rule> a acl:Authorization ; \
  oplacl:hasAccessMode oplacl:Read ; \
  acl:accessTo <urn:foobar> ; \
  acl:agent <http://www.facebook.com/john.tester> ; \
  oplacl:hasScope oplacl:PrivateGraphs .';

$.ajax('http://HOST/acl/rules?sid=' + encodeURIComponent('SID'), {
  type: 'POST',
  contentType: 'text/turtle',
  data: ruleData,
  dataType: 'text'
});
```

We use a jQuery `ajax` request to perform a `POST`. We do authentication via a URL parameter. The `dataType` should be set to `text` to prevent jQuery from trying to hopelessly detect our default `text/turtle` return type.

Example: Add a member to a group using the RESTful API:

```
var ruleData = '<#group> a foaf:Group ; \
  foaf:member <http://www.facebook.com/john.tester> .';

$.ajax('http://HOST/acl/rules/42?sid=' + encodeURIComponent('SID'), {
  type: 'PATCH',
  contentType: 'text/turtle',
  data: ruleData,
  dataType: 'text'
});
```

As before authentication information is set via a URL parameter.

Chapter 3

VAL Configuration

VAL can be configured by manipulating the triples in a set of pre-defined private graphs (see also Named Graphs Used Throughout VAL). Typically there will be user Interfaces which hide these details from the user but it is good to know the details anyway.

3.1 Page Footers

VAL's own `/sparql` integration allows to set a custom page footer. This can be used to for example show social sharing controls via Javascript commands. Each endpoint has its own configuration. The following example shows how the main `/sparql` endpoint of `http://my.openlinksw.com` can be enhanced with social sharing controls:

```
sparql
prefix oplcfg: <http://www.openlinksw.com/ontology/configuration#>
with <urn:virtuoso:val:config>
insert {
  <http://my.openlinksw.com/sparql> oplcfg:hasFooter [
    a oplcfg:HtmlSnippet ;
    oplcfg:hasHtmlBody """<script type="text/javascript"
      src="//s7.addthis.com/js/300/addthis_widget.js#pubid=xa-52ed0731782daa54"></script>
<script type="text/javascript">
  addthis.layers({
    'theme' : 'transparent',
    'share' : {
      'position' : 'right',
      'services' : 'google,linkedin,twitter,facebook,more'
    }
  });
</script>"""
  ] .
}
```

3.2 Customizing the Standard VAL Authentication Page

VAL allows to somehow customize the `authenicate.vsp` page (see also Adding VAL Support to a VSP-based Application).

3.2.1 Customize the Logos

Logos displayed on the authentication page can easily be customized per application realm. By default VAL uses the Virtuoso logo as the right image and details about the identity provider on the left.

In order to set the left and right logos for the default realm one can simply insert corresponding triples into the VAL config graph:

```
sparql
prefix oplcfg: <http://www.openlinksw.com/ontology/configuration#>
prefix oplacl: <http://www.openlinksw.com/ontology/acl#>
insert into <urn:virtuoso:val:config> {
  oplacl:DefaultRealm oplcfg:hasLeftLogo <http://path/to/logo.png> .
  oplacl:DefaultRealm oplcfg:hasRightLogo <http://path/to/another/logo.png> .
};
```

Similarly the corresponding anchors (which default to `http://www.openlinksw.com/` and `http://virtuoso.openlinksw.com/`) can be set via:

```
sparql
prefix oplcfg: <http://www.openlinksw.com/ontology/configuration#>
prefix oplacl: <http://www.openlinksw.com/ontology/acl#>
insert into <urn:virtuoso:val:config> {
  oplacl:DefaultRealm oplcfg:hasLeftAnchor <http://me.com/> .
  oplacl:DefaultRealm oplcfg:hasRightAnchor <http://me.com/coolstuff> .
};
```

3.2.2 Request Access Dialog

There are two modes to how the request access dialog is to be presented: 1. the user needs to press a button to show it (the default), or 2. the dialog is shown automatically as soon as access has been denied for an authenticated person.

This setting is tied to the application realm which means that it does not apply to any other realm.

In order to make the dialog shown automatically in the default realm one sets the following property:

```
sparql
prefix oplcfg: <http://www.openlinksw.com/ontology/configuration#>
insert into <urn:virtuoso:val:config> {
  <urn:virtuoso:val:realms:default> oplcfg:hasRequestAccessDialogMode oplcfg:SimpleRequestAccessDialog
};
```

In order to restore the default one simply deletes the configuration:

```
sparql
prefix oplcfg: <http://www.openlinksw.com/ontology/configuration#>
delete from <urn:virtuoso:val:config> {
  <urn:virtuoso:val:realms:default> oplcfg:hasRequestAccessDialogMode oplcfg:SimpleRequestAccessDialog
};
```

3.3 Customizing the ACL Graphs

The The VAL ACL Rule and Group System uses a set of named graphs to store rules, groups, and restrictions. By default VAL uses one graph per application realm and ACL resource type. It uses the default hostname (`HOST` in the example below) of the Virtuoso instance.

Example: The default graph which stores the rules in the default realm is the following:

```
http://HOST/acl/graph/rules/http%3A%2F%2Fwww.openlinksw.com%2Fontology%2Facl%23DefaultRealm
```

On firsts usage of the API to create a rule, group, or restriction this graph will be created and made private. It will then be stored in the VAL configuration using the oplacl:hasRuleDocument property:

```
oplacl:DefaultRealm oplacl:hasRuleDocument
      <http://HOST/acl/graph/rules/http%3A%2F%2Fwww.openlinksw.com%2Fontology%2Facl%23DefaultRealm> .
```

It is possible to customize these graphs (ideally before the API creates them) which might be desireable for manual ACL resource creation via SPARQL Insert. Since VAL will honor the setting above one can simply add the required triples into the VAL config graph.

Example: Given that one wants to change the rule, group, and restriction graphs for the default application realm, the following will do:

```
sparql
prefix oplacl: <http://www.openlinksw.com/ontology/acl#>
prefix oplres: <http://www.openlinksw.com/ontology/restrictions#>
with <urn:virtuoso:val:config>
insert {
  oplacl:DefaultRealm oplacl:hasRuleDocument <urn:acl:rules> ;
    oplacl:hasGroupDocument <urn:acl:groups> ;
    oplres:hasRestrictionDocument <urn:acl:restrictions> .
};
```

VAL will honor this settings and store and read all rules, groups, and restrictions from the configured graphs.

Warning

Be aware though that **VAL does not automatically migrate** rules, groups, and restrictions between graphs. This means that changing the graphs will disable existing rules, groups, and restrictions.

It is highly recommended to make these graphs private:

```
DB.DBA.RDF_GRAPH_GROUP_INS ('http://www.openlinksw.com/schemas/virtrdf#PrivateGraphs',
'urn:acl:rules');
DB.DBA.RDF_GRAPH_GROUP_INS ('http://www.openlinksw.com/schemas/virtrdf#PrivateGraphs',
'urn:acl:groups');
DB.DBA.RDF_GRAPH_GROUP_INS ('http://www.openlinksw.com/schemas/virtrdf#PrivateGraphs',
'urn:acl:restrictions');
```

Chapter 4

VAL Internals - For Developers and Power-Users

4.1 Named Graphs Used Throughout VAL

VAL uses a set of private named graphs to store all kinds of configuration and user data. This includes ACL rules and the likes. The following sections give an overview of the graphs used in VAL.

4.1.1 The VAL Configuration Graph

VAL uses one main configuration graph named `urn:virtuoso:val:config` (See also VAL.DBA.val_config↩ _graph_uri()).

This graph is typically filled manually or by UI.

4.1.2 Graphs used in the VAL ACL System

VAL's own The VAL ACL Rule and Group System uses a number of private graphs to store its data:

4.1.2.1 VAL ACL Rule Graphs

VAL's ACL system uses one private graph for rules, one for groups, and one for restrictions. Each application realm defines its own set of rules, groups, and restrictions. Thus, each realm has its own set of these three private graphs. The following list shows the default graphs which can be customized as described in Customizing the ACL Graphs. (In the following examples `HOST` refers to the default hostname of the Virtuoso instance.)

- The graph that stores ACL rules consists of a prefix `http://HOST/acl/graph/rules/` and the URL-encoded realm URI (see also VAL.DBA.val_acl_rule_graph()).
  Example: The rule graph for the default realm is `http://HOST/acl/graph/rules/http%3A%2↩ F%2Fwww.openlinksw.com%2Fontology%2Facl%23DefaultRealm`.
- The graph that stores ACL groups consists of a prefix `http://HOST/acl/graph/groups/` and the URL-encoded realm URI (see also VAL.DBA.val_acl_group_graph()).
  Example: The group graph for the default realm is `http://HOST/acl/graph/groups/http%3↩ A%2F%2Fwww.openlinksw.com%2Fontology%2Facl%23DefaultRealm`.
- The graph that stores ACL restrictions consists of a prefix `http://HOST/acl/graph/restrictions/` and the URL-encoded realm URI (see also VAL.DBA.val_restrictions_graph()).
  Example: The restrictions graph for the default realm is `http://HOST/acl/graph/restrictions/http%3↩ A%2F%2Fwww.openlinksw.com%2Fontology%2Facl%23DefaultRealm`.

4.1.2.2 VAL ACL Scheme Graph

To ensure that nobody can tamper with default access modes and the like it is important that the Openlink ACL and restriction ontologies are stored in a private trusted graph.

VAL uses the ACL schema graph `urn:virtuoso:val:acl:schema` for this purpose. It is mandatory for both the `ACL` and the `restriction` ontologies to be loaded into this graph for the VAL ACL system to work properly.

Other applications also need to copy their specific ACL scope definitions into this graph.

See also

VAL.DBA.get_acl_schema_graph ()

4.1.2.3 VAL ACL Resource Ownership Graphs

VAL defines one resource ownership graph group for each scope. The graph consists of a prefix `urn↩ :virtuoso:val:ownership:` and the URL-encoded scope URI (see also VAL.DBA.ownership_graph_group ()).
Example: The ownership graph group for the private graph scope is `urn:virtuoso:val:ownership↩ :http%3A%2F%2Fwww.openlinksw.com%2Fontology%2Facl%23PrivateGraphsScope`.

4.1.3 VAL owl:sameAs graph

VAL uses private graph `urn:virtuoso:val:online:accounts` (see also VAL.DBA.val_owl_sameas_↩ graph()) to store `owl:sameAs` relations for all service ids which are considered to identify the same person. See also VAL.DBA.update_user_online_mapping().

Chapter 5

VAL Tutorials

5.1 Adding VAL Support to a VSP-based Application

VAL provides the means to easily add authentication and ACL support to existing or new vsp-based applications. This tutorial shows the three main steps to add authentication and ACL protection to an application. We are using `curi` - the Compressed URI Service as an example.

For `curi` we want to add login information, a means for the user to logout, and ACLs to protect the service.

5.1.1 Check for existing authentication information

The first and simplest step is to check if the user already provided authentication information as supported by VAL. This can simply be achieved by calling VAL.DBA.get_authentication_details_for_connection() at the top of the `vsp` page:

```
declare val_serviceId, val_sid, val_realm, val_uname varchar;
declare val_isRealUser integer;
declare val_cert any;

-- Important since "realm" is an "inout" parameter!
val_realm := null;

VAL.DBA.get_authentication_details_for_connection (
  sid=>val_sid,
  serviceId=>val_serviceId,
  uname=>val_uname,
  isRealUser=>val_isRealUser,
  realm=>val_realm,
  cert=>val_cert);
```

Note

> VAL.DBA.get_authentication_details_for_connection() will honor a non-null `realm` parameter and only return authentication data for the given realm. Additionally it will honor the `app_realm` setting in the virtual dir serving the page in question. Thus, there are basically two ways to define the realm for an application: 1. Set it in the virtual dir, and 2. Force it manually via the `realm` parameter.

After the call to VAL.DBA.get_authentication_details_for_connection() the application can use the information. The most important one is the value of `val_serviceId` which defines who is authenticated. If it is `null` then the user has not authenticated yet.

5.1.2 Adding Login and Logout Links

VAL provides an authentication and a logout page to support the most simple login and logout links possible. Given that the application page is stored in `pageUrl` the following links can be used:

```
<a href="/val/authenticate.vsp?returnto=<?/pageUrl?>">Login</a>
<a href="/val/logout.vsp?returnto=<?/pageUrl?>">Logout</a>
```

However, in our case a dedicated login page is more desirable since it allows to configure certain aspects of `authenticate.vsp`. Thus, we create a new page `login.vsp` with the following content (or at least parts of it):

```
<?vsp
  connection_set ('__val_auth_page__', '/c/login.vsp');
  connection_set ('__val_req_res_label__', 'Compressed URI Service');
  connection_set ('__val_oauth_scope__', 'profile');
?>
<?include /DAV/VAD/val/authenticate.vsp ?>
```

The settings should be obvious:

- `__val_auth_page__` We tell `authenticate.vsp` to use `login.vsp` instead of its own URL for all login links.
- `__val_req_res_label__` A custom label for the login dialog to tell the user which service they log into.
- `__val_oauth_scope__` The optional OAuth scope to use (`basic`, `profile`, or `dav`). This is only of interest for applications that reuse the created OAuth sessions for additional API calls to the 3rd-party service.

So we end up with code for creating a login/logout box like the following:

```
if (val_serviceId is not null) {
  http (sprintf ('Logged in as %s', val_serviceId));
  http (sprintf ('<a href="/val/logout.vsp?returnto=%U">Logout</a>', pageUrl));
}
else {
  http (sprintf ('<a href="login.vsp?returnto=%U">Login</a>', pageUrl));
}
```

Once the user authenticates, they will be redirected to the `pageUrl` with a newly created `sid` cookie. the logout page will remove that cookie.

Tip: A slightly nicer logged in message with link can be created with code like the following which makes use of the two utility procedures VAL.DBA.get_profile_url() and VAL.DBA.get_profile_name():

```
declare x, n varchar;
x := VAL.DBA.get_profile_url (val_serviceId);
n := coalesce (VAL.DBA.get_profile_name (val_serviceId), val_serviceId);
if (not x is null)
  http (sprintf ('<a href="%s">%V</a>', x, n));
else
  http (n);
```

Warning

Be aware that WebID logout is not always possible as it requires a redirect to the non-ssl protected application page.

5.1.3 40x Pages

A typical situation for authentication-enabled applications is forcing the user to authenticate. Ideally this is done via `40x` page options in the virtual directory in combination with VAL's `authenticate.vsp` page (which is also used for login links). One simply creates a new file `40x.vsp` which has the following content:

```
<?vsp
  connection_set ('__val_req_res__', 'urn:virtuoso:access:curi');
  connection_set ('__val_req_acl_scope__', 'urn:virtuoso:val:scopes:curi');
  connection_set ('__val_req_res_label__', 'Compressed URI Service');
  connection_set ('__val_auth_page__', '/c/login.vsp');
  if (isstring (http_param ('error.msg')))
    connection_set ('__val_err_msg__', http_param ('error.msg'));
?>
<?include /DAV/VAD/val/authenticate.vsp ?>
```

`authenticate.vsp` can be configured via a set of connection settings:

- __val_req_res__ The resource which is protected, ie which requires the login. This is only used to retrieve ownership information for the "request access" dialog that authenticate.vsp will show if access was denied. This will default to the returnto URL if not provided, and should that also be null (as is the case if authenticate.vsp is used as 40x_page) then the requested URL will be used.
- __val_req_acl_scope__ The ACL scope in which the above resource is protected. This is only used to retrieve ownership information for the "request access" dialog that authenticate.vsp will show if access was denied. If not given, then no "request access" dialog is shown.
- __val_req_res_label__ An optional label for the login dialog showing the user for which service they are authenticating.
- __val_auth_page__ We tell authenticate.vsp to use our custom page login.vsp instead of its own URL for all login links.
- __val_err_msg__ An error message indicating any kind of error. This should be set to http_param ('error.msg') for the simple reason that Virtuoso does clear the http params before processing the 40x page.

This page will be used as 40x page in the virtual directory configuration:

```
DB.DBA.VHOST_DEFINE (
  lpath=>'/c',
  ppath=>'/DAV/VAD/c_uri/',
  is_dav=>1,
  vsp_user=>'CURI',
  ses_vars=>0,
  opts=>vector (
    'url_rewrite', 'c_uri_lst',
    '401_page', '40x.vsp',
    '403_page', '40x.vsp'),
  is_default_host=>0
);
```

Then the application can raise a permission denied error as shown in the following example:

```
if (val_serviceId is null) {
  http_status_set(401);
}
else {
  connection_set ('__val_denied_service_id__', val_serviceId);
  http_status_set(403);
}

return ";
```

If val_serviceId is null then the user has not logged in and the application simply requests that they do. Otherwise 403 indicates that permission was denied to the authenticate user. The authenticated has to be communicated to authenticate.vsp via the __val_denied_service_id__ connection setting.

5.1.4 Using ACL Rules to Protect A Web Service

In 40x Pages we saw how to use authenticate.vsp as a 40x_page. Now we will add ACL protection to the curi service and put the new 40x_page to use.

We want to be able to grant people the right to create new compressed URIs and others the right to read these. To that end we define a new scope urn:virtuoso:val:scopes:curi which is only used for curi and a virtual resource URI which is used to grant permissions: urn:virtuoso:access:curi These URIs are arbitrary, they simply follow a random scheme to be easily recognizable. In theory they could be any URI one wanted to use.

VAL makes use of scope definitions to get default access modes for disabled scopes (the default). Thus we start by defining our new scope in the corresponding VAL acl schema graph (Hint: standard scopes for DAV, etc. are defined in the OpenLink ACL ontology, example: oplacl:Dav):

```
sparql
prefix acl: <http://www.w3.org/ns/auth/acl#>
prefix oplacl: <http://www.openlinksw.com/ontology/acl#>
insert into <urn:virtuoso:val:acl:schema> {
  <urn:virtuoso:val:scopes:curi> a oplacl:Scope ;
    rdfs:label "Compressed URIs" ;
    rdfs:comment """SQL ACL scope which contains all ACL rules granting permission to create and read
      compressed URIs. By default anyone can fully use the service.""" ;
    oplacl:hasApplicableAccess oplacl:Read, oplacl:Write ;
    oplacl:hasDefaultAccess oplacl:Read, oplacl:Write .
};
```

The most important part is `oplacl:hasDefaultAccess` which defines the access modes used in case ACL evaluation has not been enabled for the `curi` scope. In this case everyone is allowed to create and read compressed URIs.

Now at the top of the `create.vsp` page which allows to create new compressed URIs we add the following ACL check (after the code from 40x Pages):

```
if (not val_isRealUser or not VAL.DBA.is_admin_user (val_uname)) {
  if (not VAL.DBA.check_access_mode_for_resource (
        serviceId=>val_serviceId,
        resource=>'urn:virtuoso:access:curi',
        realm=>val_realm,
        scope=>'urn:virtuoso:val:scopes:curi',
        mode=>VAL.DBA.oplacl_iri ('Write'),
        webidGraph=>val_webidGraph,
        certificate=>val_cert,
        honorScopeState=>1)
  ) {
    connection_set ('__val_denied_service_id__', val_serviceId);
    connection_set ('__val_req_acl_mode__', VAL.DBA.oplacl_iri ('Write'));

    if (val_serviceId is null)
      http_status_set(401);
    else
      http_status_set(403);

    return ";
  }
}
```

Some of this code we already know from before. But the big first part is new. First we check if we are logged in as an admin user. VAL provides us with the convinience procedure VAL.DBA.is_admin_user() for that. Of course only "real" users, ie. SQL users, can be administrators of the V instance. In case no admin credentials were provided we continue with the ACL check using VAL.DBA.check_access_mode_for_resource() which allows to check for exactly one mode on one resource for one service id. Here we use all the details that were provided by VAL.DBA.get_↩ authentication_details_for_connection() and combine them with the resource and scope URIs we defined above.

Since we want to create a compressed URI we use the `oplacl:Write` access mode. Should no ACL exist which grants access we continue to raise a `40x` error. But before we do that we set two more variables:

- `__val_denied_service_id__` This is important as it allows `authenticate.vsp` to know that access has been denied to a certain person and the user should be asked to login again. Without this setting, `authenticate.vsp` would simply return to the `returnto` URL if authentication information could be found. This would result in an endless loop. Should no authentication information exist yet then `authenticate.vsp` will simply ask for it.
- `__val_req_acl_mode__` Like the resource and the scope settings above the mode is only used for the "request access" dialog. It allows `authenticate.vsp` to create a more detailed access request message to the resource owner.

Finally we add the same code to the `get.vsp` page which handles the conversion of comressed to uncompressed URIs. The only difference is the access mode:

```
if (not val_isRealUser or not VAL.DBA.is_admin_user (val_uname)) {
  if (not VAL.DBA.check_access_mode_for_resource (
        serviceId=>val_serviceId,
        resource=>'urn:virtuoso:access:curi',
        realm=>val_realm,
        scope=>'urn:virtuoso:val:scopes:curi',
        mode=>VAL.DBA.oplacl_iri ('Read'),
        webidGraph=>val_webidGraph,
        certificate=>val_cert,
        honorScopeState=>1)
  ) {
    connection_set ('__val_denied_service_id__', val_serviceId);
    connection_set ('__val_req_acl_mode__', VAL.DBA.oplacl_iri ('Read'));

    if (val_serviceId is null)
      http_status_set(401);
    else
      http_status_set(403);

    return '';
  }
}
```

5.1.5 The Request Access Dialog

`authenticate.vsp` provides a simple dialog through which users can request access to a certain resource, should it have been denied. This dialog is shown by `authenticate.vsp` if the following conditions hold true:

- The user has been denied access, i.e. `__val_denied_service_id__` is `non-null` (See 40x Pages for details).
- The owner of the resource access has been denied to can be determined. This means `__val_req_res↩__` has to be `non-null` and an owner has to be set (DAV resources are handled as special cases, for every other resource see VAL.DBA.set_resource_ownership(), VAL.DBA.add_ownership_graph() and friends.)
- VAL has a means to contact the owner. That means an email address has to be known (see also VAL.↩DBA.email_address_for_service_id()) and the instance's sendmail configuration has to be valid (see also VAL.DBA.smtp_server_available()).

Once these conditions are fulfilled then the user has the option to write a message to the owner of the resource, requesting to grant them access.

5.1.6 Running an Application Under a Specific SQL User Account

In the case of `curi` the vsp pages are not executed as `dba` but using the dedicated account `CURI` which improves security and is generally recommended. However, since most of the internal VAL API procedures require special permissions this user needs to be granted the `VAL_AUTH` and `VAL_ACL` roles to be able to execute:

```
grant VAL_AUTH to CURI;
grant VAL_ACL to CURI;
```

(The API documentation contains hints about which role grants the right to execute a specific procedure in said procedure's documentation.)

5.2 Enforcing SPARQL ACL Rules

As explained in SPARQL ACL Rules - Defining access to private graphs VAL handles SPARQL ACL rules as a special case, using the fixed rule scope oplacl:PrivateGraphsScope (VAL.DBA.get_sparql_scope()) and a graph security callback. The following example shows how all the different pieces of VAL can be used to perform ACL-protected SPARQL queries (as done by `sparql.vsp`).

Imagine the actual SPARQL query is stored in variable `query`.

First we get the authentication information for the current connection. It is important to set `val_realm` to `null` because it is an `inout` variable which has to be tested against `null` in the procedure.

```
declare val_serviceId, val_sid, val_realm, val_uname, val_webidGraph varchar;
declare val_isRealUser integer;
declare val_cert any;

-- Set the inout variable to null since we want to get the used realm back
val_realm := null;

-- Use a tmp graph for the WebID profile
val_webidGraph := uuid();

-- Fetch the authentication details
VAL.DBA.get_authentication_details_for_connection (
  sid=>val_sid,
  serviceId=>val_serviceId,
  uname=>val_uname,
  isRealUser=>val_isRealUser,
  realm=>val_realm,
  cert=>val_cert,
  webidGraph=>val_webidGraph);
```

We use `sid` as name for the cookie which stored the session id. This is the default but can be set to any value required. (However, be aware that `authenticate.vsp` currently does not allow to change this.)

After the call to VAL.DBA.get_authentication_details_for_connection() we know if a user is logged in or not and can act accordingly. One might either want to show a login link or simply proceed to execute the query as user `nobody` (which means that only public ACL rules apply).

```
-- Set the effective user (when using WS.WS."/!sparql/")
connection_set ('SPARQLUserId', 'VAL_SPARQL_ADMIN');

-- Set the effective user (when directly executing the query)
set_user_id ('VAL_SPARQL_ADMIN');

-- Set the application realm to use in the sparql graph security callback
connection_set ('val_sparql_rule_realm', val_realm);

-- Let the callback know about the mapped SQL user for system permissions
if (not val_uname is null)
  connection_set ('val_sparql_uname', val_uname);

-- let the callback know about the webid tmp graph
connection_set ('val_sparql_webid_graph', val_webidGraph);

-- inject the graph security callback function into the query
query := sprintf ('define sql:gs-app-callback "VAL_SPARQL_PERMS" define sql:gs-app-uid "%s" ', coalesce
      (val_serviceId, 'nobody')) || query;
```

As recommended we execute the query as user `VAL_SPARQL_ADMIN` to get full access to the entire triple store. We then use the callback to restrict that access according to the ACL rules setup in the current realm. We do not use `dba` for security reasons. `VAL_SPARQL_ADMIN` is a special user which has full access to all public and private graphs. The permissions reported by the callback include not only the *implied* permissions set by VAL ACL rules, but also the *physical* graph permissions set by `RDF_USER_PERMS_SET`.

Finally we can execute the query and clear the temporary WebID graph:

```
sparql clear graph ?:val_webidGraph;
```

Chapter 6

Supported Authentication Services and Methods

6.1 Introduction

VAL supports a variety of authentication services via VAL.DBA.thirdparty_authentication_url() and VAL.DBA.↩ digest_authentication(). In addition VAL.DBA.authentication_details_for_connection() supports several means of authenticating.

The following gives an overview of these services and examples of the identifiers used.

6.2 Supported 3rd Party Services

VAL supports a wide range of authentication methods and services. Authentication with any of those methods always yields a personal URI of some kind. This URI is referred to as a NetID, In other words: a NetID is any URI that identifies a person on the network.

The following list shows the different types of authentication services that can be used to authenticate via VAL. Each authentication session resolves to one unique service ID or NetID, i.e. a personal URI. Examples are included in the list.

Service Type	Identifier used	NetID Example
`facebook`	The profile URL	`http://www.facebook.com/sebastian.trug`
`twitter`	The profile URL	`http://twitter.com/tmptrueg`
`linkedin`	The profile URL	`https://www.linkedin.com/in/trueg`
`windowslive`	The profile URL	`http://profile.live.com/cid-7a6b1666d21a866b/`
`google`	The Google Plus URL	`https://plus.google.com/100942621970840297012`
`wordpress`	The main blog URL	`http://trueg.wordpress.com`
`disqus`	The profile URL	`http://disqus.com/strueg/`
`instagram`	An acct: URL	acct: `176087504@instagram.com`
`yahoo`	The profile URL	`http://profile.yahoo.com/S4KF6WXGWUBRV74↩ G4TD6GOPCAE`
`tumblr`	The profile URL	`http://trueg.tumblr.com/`
`bitly`	The profile URL	`http://bitly.com/u/webods`
`dropbox`	An acct: URL	acct:trueg@dropbox.com

Service Type	Identifier used	NetID Example
flickr	The profile URL	http://www.flickr.com/people/91384569@N06/
bitly	The profile URL	http://bitly.com/u/strueg
foursquare	An acct: URL	acct:39809951@foursquare.com
github	The profile URL	https://api.github.com/users/trueg
meetup	The profile URL	http://www.meetup.com/members/73346552
salesforce	The profile URL	https://login.salesforce.com/id/00↩ Db0000000J2hSEAS/005b0000000QUONAA4
boxnet	An acct: URL	acct:189578964@box.com
xing	A profile URL	https://www.xing.com/profile/Max_Mustermann
beatport	An acct: URL	acct:trueg@beatport.com
amazon	An acct: URL	acct:amzn1.account.AGTL7CUF4HT3FTBFX3JWF7↩ QFGKFG@amazon.com
soundcloud	A profile URL	http://soundcloud.com/user922973331
webid	The personal URI	http://web.ods.openlinksw.↩ com/dataspace/person/trueg#this
openid	The OpenID URL	

6.3 Supported Authentication Methods

6.3.1 Session IDs

Authenticating via `authenticate.vsp` (which internally uses VAL.DBA.thirdparty_authentication_url(), and VAL.DBA.digest_authentication()) results in a new session ID which is stored in a new cookie named `"sid"`. This cookie, as long as it is valid, can be used to authenticate to any endpoint using VAL.DBA.authentication_↩ details_for_connection().

Alternatively the session ID can be provided in a url parameter by the same name: `"sid"`.

6.3.2 Basic PKI

In this authentication scenario, a standard X.509 client certificate is all that's required. VAL will use the certificate's fingerprint as the service ID.

Example: Given a certificate fingerprint DC:18:68:D3:4F:9A:08:71:38:4B:D8:B2:74:3E:BE:87 the service ID would be cert:DC:18:68:D3:4F:9A:08:71:38:4B:D8:B2:74:3E:BE:87.

VAL also supports a custom HTTP header `X-Application-Realm` which allows to set the application realm when authenticating. This is particular useful for API calls which have different results based on the realm. A typical example is the VAL Public ACL HTTP API.

6.3.3 WebID+TLS

VAL supports authentication via WebID+TLS. If a client certificate is sent with the request that contains a valid WebID in its SAN field, then that WebID will be used as a valid service id.

VAL also supports a custom HTTP header `X-Application-Realm` which allows to set the application realm when authenticating. This is particular useful for API calls which have different results based on the realm. A typical example is the VAL Public ACL HTTP API.

6.3.4 HTTP Authentication

VAL supports plain HTTP authentication for "real" SQL users of the Virtuoso instance. This is useful for tests and actions performed via curl and friends:

```
$ curl -u dba:dba -H "Accept:text/turtle" "http://HOST/acl/rules"
```

Chapter 7

Topic Index

7.1 Topics

Here is a list of all topics with brief descriptions:

Chapter 8

Topic Documentation

8.1 VAL Public ACL HTTP API

Tip: Maybe you would rather use the The RESTful ACL API.

The public HTTP API provides API functions to manage rules and groups. It is entirely turtle-based, meaning that input and output is in turtle format. By default VAL installs a virtual directory at `/val/api` which means the API calls will be available as `http://HOST/val/api/acl.rule.new` and so on. All VAL-supported authentication mechanisms are supported. This includes session ids, OAuth, HTTP digest auth, and `WebID`. Be aware, though, that only the session id carries an application realm at the moment. All other authentication mechanisms will result in usage of the default realm `oplacl:DefaultRealm`.

8.1.1 The RESTful ACL API

The RESTful ACL API allows to create, update, delete, and list rules and groups using typical RESTful patterns. Two base URLs serve as entry points:

- `http://HOST/acl/rules` - Will list existing rules via a `GET` operation and allows to create new rules by `POST`ing them.
- `http://HOST/acl/groups` - Will list existing groups via a `GET` operation and allows to create new groups by `POST`ing them.
- `http://HOST/acl/restrictions` - Will list existing restrictions via a `GET` operation and allows to create new restrictions by `POST`ing them.

Existing rules, groups and restrictions can be manipulated directly via their URLs. An HTTP `DELETE` will remove them; an HTTP `PATCH` allows to add members or conditions to groups and modes to rules; an HTTP `PUT` allows to overwrite entire groups, rules or restrictions (excluding the realm which cannot be changed).

The APIs prominent data exchange format is `turtle`.

For usage examples see ACL API Examples Using Curl.

8.2 VAL Internal ACL API

Warning

Please do not use the internal ACL API if you could also use The RESTful ACL API or VAL Public ACL HTTP API!

The internal ACL API allows vsp-based applications to manage ACL rules. However, this should only be used if the HTTP API is for some reason not sufficient.

8.3 VAL Internal ACL Utility API

The procedures in the utility API can be used in applications to enforce the ACL rules and to maintain Resource Ownership. Most notably VAL.DBA.val_prepare_sparql_permissions_for_query() needs to be called before each SPARQL query to ensure proper permissions.

In contrast to the procedures in VAL Internal ACL API the usage of these procedures is encouraged and often necessary.

8.4 VAL Authentication API Tools

The Authentication Tools API contains a set of convinience procedures which make working with VAL is vsp pages much simpler.

8.5 VAL Authentication API

The VAL Authentication API provides a set of procedures to easily integrate authentication functionality into vsp pages.

Index