Xaraya
Request for Comments: 0030
Category: Best Current Practice

M. Lutolf
Xaraya Development Group
November 2002

Security System

Status of this Memo

This document specifies a Xaraya Best Current Practices for the Xaraya Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This RFC presents a proposal for the Xaraya security system. The main difference with respect to the existing security system is that privileges are defined as objects independent of users and groups. As a corollary, the structure of users and groups is also revisited.

Table of Contents

1. Definition of terms

To avoid confusion before specifying anything, here is the list of terms we use in relation to the security system.

- Role: refers to a named set of 1 or more users. A role can correspond to a group or a user. Role names are unique.

Example: the site administrator or the group of registered users are both roles.

- Privilege: refers to an object that can grant a particular kind of access to a module component (see below).

Example: the ability to see the front page, or its login block is a privilege.

- Module: refers to a functional unit of a Xaraya site.
- Component: refers to a group of resources that can be protected with the security system. Components are defined by the modules they reside in. Components can be visually discrete resources, such as blocks or web pages, or they can be functionality resources, such as the ability to add or edit items.
- Instance: refers to a specific instance of a component.

Example: the login block on the front page of a Xaraya site is an instance of the Block component in the Roles module. An administrator might choose to create more than one such instance e.g. in a multilingual site.

- Mask: refers to a special kind of privilege used to check other privileges. Each mask protects one or more components. When a mask is encountered it is checked against the user's privileges to see whether access to the component in question is granted.
- Access level: refers to the type of access granted by a privilege or required by a mask.

2. Introduction

- The main points of this RFC are:
  - Privileges are defined as objects that can be manipulated independently of their specific assignments to roles.
  - Users and groups are merged in the implementation into a single class of objects called roles.
  - Both roles and privileges can have unlimited levels of subroles and subprivileges respectively.
  - Components, i.e. objects that can have privileges assigned to them through their registered mask.
- Benefits:
  - Privileges can be manipulated independently of roles. We can create groups of privileges, adding or deleting their components.
  - We can then assign (compound) privileges to roles in a single statement.
  - Registering components allows us to make visible to the user the available privileges to be set.
- This RFC does not cover:
  - The issue of translating the current system of using regex for privileges into the privileges of the proposed scheme.
  - Whether and how regex should be allowed in the new UI.
- Note: This RFC has borrowed generously from the concepts underlying the Java packages java,security and java.security.acl. These can be found at: http://java.sun.com/j2se/1.3/docs/api[1]

[1] http://java.sun.com/j2se/1.3/docs/api

3. Concepts of the Xaraya Security System

3.1 Roles

Roles is a generalization of the Users and Groups concepts in PostNuke. In Xaraya both Users and Groups of users are implemented as roles. As in PostNuke, roles that are Users can be members of other roles that are Groups. The difference to the PostNuke concepts is that there is no limit to the levels of nesting available. In addition, a role can be a member of any number of other roles.

The roles in Xaraya are arranged in a hierarchy or tree, and every role defined must be a member of at least one already existing role. At the top of the hierarchy is a role called "Everybody".

Note: Contrary to the PostNuke permissions system, the order in which multiple parents of a role are defined is not relevant for the way privileges are applied and has no impact on whether access is granted or denied. Specifically, there is no "first encountered wins" rule in Xaraya. ALL the privileges available will be checked for access.

The following rules apply to the roles hierarchy:

1. Roles defined as Users cannot themselves have children. (i.e. a role which is a user is a leaf of the roles tree)
2. A role cannot be a parent of one of its ancestors in the hierarchy; obviously to prevent endless loops. A role also cannot be its own child.
3. A role cannot have another role as a child more than once. (This is not strictly necessary, but it avoids confusion.) Note, however, that it is possible for a role to be a child and simultaneously a grandchild of another role, as the following example shows.

```
Role A              Big Boss
|                   |
|--Role B           |--Small Boss
|  |                |  |
|  |--Role C        |  |--John Doe
|                   |
|--Role C           |--John Doe
```

John both reports to "Big Boss" and "Small Boss" and as such has two roles in the system.

In the user interface we use the term "User" for roles which have one member and "Group" for roles which have multiple members. Conceptually however there is no difference between the two, both are roles. Since one can expect numerically many more Users than Groups or, in other terms, more leaves then nodes, the former are not displayed by default. Rather, one can specifically request to see the Users belonging to a given Group.

For convenience a number of roles are created at initialization time. They are:

```
Everybody           (Group)
|
|--Administrators   (Group)
|  |
|  |--Admin         (User)
|
|--Users            (Group)
|
|--Anonymous        (User)
```

Everybody: This is the root of the roles tree and is defined as a group that contains all other subgroups and users.

Administrators: This is a group that contains those users that can do anything in the system, a sort of superuser. The Administrators group contains one such user, the Admin. The Admin is the default user logged on when Xaraya is installed.

Users: This is a group that represents users of the site. By default new users that register on the site become members of this group. This default can be changed by modifying the configuration settings of the Base module.

Anonymous: This is a user that represents users of the site that are not logged in.

Myself: This is a special "meta-user" that represents the current user. Myself can be used to create special privileges that allow, for instance, only the author of an article to modify the article.

3.2 Privileges

The Xaraya concept of privileges has a structure similar to privileges in PostNuke. A privilege is an object that grants a particular type of access to a resource. Every privilege incorporates:

- A reference to the Module it applies to. In addition to the list of available modules, a privilege can also apply to "All".
- A reference to the Component within the Module it applies to. In addition to the list of available components of a module, a privilege can also apply to "All".
- A reference to the Instance of the Component it applies to. In addition to the list of available instances, a privilege can also apply to "All".
- A privileges Level it refers to. The levels have the usual.values of ACCESS_NONE to ACCESS_ADMIN (numeric 0 - 800).

In addition, each privilege also has:

- A name which identifies it. The name is used as a shorthand reference to the privilege in the UI and must be unique. It is good practice to use names that help understand the rights a privilege grants, such as "ReadArticleItems" or "DeleteExamplesBlocks".
- A description that contains detail information about the privilege. The description is meant as documentation to developers and is not referenced in the UI. It is not mandatory.

Similar to roles, privileges can be composed of other privileges. In addition to the attributes described above a privilege can have any number of subprivileges. A privilege can also be a part of any number of "parent" privileges. However, privileges do not live in a single hierarchy. Instead, the UI presents all the privileges in a list, with each privilege containing its subprivileges in a separate list, as the following example shows:

```
NoPrivileges                a privilege with no subprivileges

Administration              another privilege with no subprivileges

CasualAccess                a privilege with 2 subprivileges,
 contains:
   ViewRolesBlock           one of which itself contains
    contains:
      ViewLoginBlock        2 subprivileges.
      ViewOnlineBlock
   ViewThemes
```

N.B.: Privileges can be defined as empty containers, i.e. not themselves granting any rights. This is useful if they only serve to group other privileges. In the example above the privilege CasualAccess is an empty container that holds its 2 subprivileges.

It's important to understand that grouping privileges like this does *not* imply that "parent" privileges are in any way "stronger" than "children". Making privileges subprivileges to other privileges is just a convenient way of grouping them, so that bundles of them can be assigned at a time.

In the above example note that although the names are suggestive of certain behaviour there is no fixed naming rule. You can assign any names you want as long as they are unique within their module.

Furthermore there is no rule for how to group privileges, for instance by module. A good administrator will structure his privileges according to the roles he wants to assign them to, rather than by component or level.

In Xaraya privileges are defined without reference to any group or user. They are objects with no relevance until they are assigned to one or more roles. As explained below, assigning a privilege to a role also assigns ALL the subprivileges of that privilege to the role.

For convenience a number of privileges are created with each installer configuration at initialization time. Examples of these are:

- "Administration" is the privilege representing the highest level of access to all Xaraya modules, i.e. the ability to do anything. It has the attributes:

```
Realm     Module    Component    Instance    Level
--------- --------- ------------ ----------- ------------
All       All       All          All         ACCESS_ADMIN
```

- "CasualAccess" is the privilege defined for the unregistered user Anonymous. Unregistered users initially have access only to the front page of the site and the login block.

3.3 Access Levels

As mentioned above Xaraya uses the access levels originally defined in PostNuke. These are:

```
Name              Level
--------------    -----
ACCESS_NONE          0
ACCESS_OVERVIEW    100
ACCESS_READ        200
ACCESS_COMMENT     300
ACCESS_MODERATE    400
ACCESS_EDIT        500
ACCESS_ADD         600
ACCESS_DELETE      700
ACCESS_ADMIN       800
```

The levels are cumulative. A given level implies the right to all the levels below it. The right to DELETE implies the rights to ADD, EDIT, MODERATE etc. An exception is ACCESS_NONE, which can override any other access right.

3.4 Irreducible Sets

As noted above, when two privileges are defined on the same module, component and instance the one with the higher access level includes the right to the one with the lower level. We say that the privilege with the higher access level implies the other. In the following example two privileges are defined on the Examples module:

```
            Name              Realm     Module    Component    Instance    Level
            ----------------- --------- --------- ------------ -----------
------------
            ReadExampleBlock  All       Examples  Block        All
ACCESS_READ
            EditExampleBlock  All       Examples  Block        All
ACCESS_EDIT
```

The privilege EditExamples implies ReadExamples.

Note that two privileges A and B are considered equal (identical) if A implies B and B implies A.

In the following example, however:

```
            Name              Realm     Module    Component    Instance    Level
            ----------------- --------- --------- ------------ -----------
------------
            ReadExampleBlock  All       Examples  Block        All
ACCESS_READ
            EditArticleBlock  All       Articles  Block        All
```

```
ACCESS_EDIT
            EditArticleItem   1         Articles  Item         All
ACCESS_EDIT
```

none of the privileges imply any of the others, as they refer to different modules and/or realms. We call such privileges disjoint. (NOTE: This example contains a Realm which is a feature which will not be used in the initial implementation.)

Definition: An Irreducible Set of privileges is a set in which all the privileges are disjoint.

3.5 Winnowing

Definition: Winnowing is the process by which the Xaraya privileges system creates irreducible sets of privileges.

A privilege accumulates the attributes of its children. As mentioned above since each privilege refers to a single realm/module/component/instance combination, you create more complex privilege-schemes by successively adding privileges to other privileges. During a security check all the component privileges in the tree will be taken into account.

However, in any given tree not all the privileges will necessarily be relevant. Take the following example:

```
            ReadAll
            |
            |--DeleteExamples
            |  |
            |  |--AddExamples
            |
            |--EditArticles
            |
            |--AddArticles

            with the following definitions:

            Name                Realm     Module    Component    Instance    Level
            ------------------- --------- --------- ------------ -----------
------------
            1. ReadAll          All       All       All          All
ACCESS_READ
            2. DeleteExamples   All       Examples  All          All
ACCESS_DELETE
            3. AddExamples      All       Examples  All          All
ACCESS_ADD
            4. EditArticles     All       Articles  All          All
ACCESS_EDIT
            5. AddArticles      All       Articles  All          All
ACCESS_ADD
```

It can be seen that, with the definitions given above, 3 implies 2 and 5 implies 4. Therefore when comparing all the privileges in the set, privileges 2 and 4 are not relevant, because they are superceded by 3 and 5 respectively.

During the winnowing process the privileges system compares each privilege in a tree with all the others in the same tree and discards those privileges implied by others. In particular duplicate privileges are removed. In the example above winnowing would leaves us with the following privileges:

```
            ReadAll
            DeleteExamples
            AddArticles
```

As can be seen, all the privileges left in the example above after winnowing are disjoint. The set is irreducible.

Note also that the system doesn't care what the tree looks like. The same result would be gotten with the following tree, among others:

```
ReadAll
|
|--DeleteExamples
|
|--AddExamples
|
|--EditArticles
|
|--AddArticles
```

[NOTE: MrB: This is somewhat counterintuitive. I would expect the shape of the tree to have influence. I understand it technically doesn't, but by representing it as a tree the user expects AddExamples to be "a part of" DeleteExamples, whatever that means to him. The difference between "defining" the tree and "checking the tree is a concept which is mentally merged for the user.]

3.6 Assigning Privileges

In order for the privileges system to do something useful, privileges need to apply for users and groups. This is done by assigning privileges to roles in the UI. Theoretically any privilege can be assigned to any role.

Suppose I have a role FOO and the tree of privileges in the example above. The operation:

"assign ReadAll to FOO"

will assign all 5 of the privilege in the tree to FOO. (Note the notation for assignment shown here is only for convenience; the privileges UI is graphical)

However, note that the operation:

"assign DeleteExamples to FOO"

will give a different result depending on the shape of the tree, as the following examples show:

```
ReadAll                  "assign DeleteExamples to FOO"
|
|--DeleteExamples
   |                     Privileges assigned: DeleteExamples
   |--AddExamples                             AddExamples
      |                                       EditArticles
      |--EditArticles                         AddArticles
      |
      |--AddArticles


ReadAll                     "assign DeleteExamples to FOO"
|
|--DeleteExamples            Privileges assigned: DeleteExamples
|  |                                              AddExamples
|  |--AddExamples
|
|--EditArticles
|
|--AddArticles
```

3.7 Default Assignments

For convenience a number of assignments are made at initialization time.

For instance, the role Administrators is assigned the privilege Administration. In other words, members of the Administrators group can do anything.

In addition, a special privilege called GeneralLock is also assigned to the roles created at installation. GeneralLock contains privileges that stop you from changing or deleting basic roles and privileges such as the site Admin or Everybody. Removing or altering GeneralLock makes it possible to make such changes. This should only be done in special cases and with an understanding of how the system works, however, as such changes can damage the site in ways difficult to recover from.

3.8 Inheriting Privileges

A privilege assigned to a role is automatically inherited by all the role's descendants. For example, using the previous example:

```
            ReadAll                   "assign ReadAll to FOO"
            |
            |--DeleteExamples
            |  |                       Privileges assigned: DeleteExamples
            |  |--AddExamples                                AddExamples
            |                                                EditArticles
            |--EditArticles                                  AddArticles
            |                                               and ReadAll (if it is
not empty)
            |--AddArticles
```

Role FOO will have 5 privileges assigned to him. However the same privileges will also be assigned to a child of FOO:

```
            FOO
            |
            |--BAR
```

We say that BAR has inherited the privileges of FOO. Note that BAR may also have privileges assigned to it. The UI of the privileges system will show for each role which privileges have been assigned and which inherited.

It's pretty apparent that if an inherited and an assigned privilege are disjoint, then nothing will happen. Essentially the two privileges have nothing to do with each other. But what happens when one of them implies the other? To wit:

```
            FOO                 has assigned DeleteExamples
            |
            |--BAR              has assigned ReadExamples

            where

            Name                Realm     Module    Component    Instance    Level
            ------------------- --------- --------- ------------ -----------
------------
            2. DeleteExamples   All       Examples  All          All
ACCESS_DELETE
            3. ReadExamples     All       Examples  All          All
ACCESS_READ
```

In this case the relevant privilege for BAR would be ReadExamples, even though its access level is lower, because

Rule: For privileges that are not disjoint, those of the children take precendence over those of the parents. We say that BAR's privilege "trumps" that of his parent FOO.

3.9 Masks

A mask is a special type of privilege in the Xaraya security system. Each mask refers to one or more security checks in the code. When a security check is encountered, the system gets the relevant mask and compares it to the privileges of the user. A green light is given if the mask is implied by one of the user's privileges. Otherwise an exception is thrown.

The structure of a Schema corresponds to that of a privilege. For example the following Schema

```
            Name          Realm     Module    Component    Instance    Level
            ------------- --------- --------- ------------ -----------
------------
            ReadExamples  All       Examples  All          All         ACCESS_READ
```

requires that a user has a privilege assigned with at least ACCESS_READ in order to pass a given security check.

3.10 How the System checks Privileges

When a security check is encountered in the code, the privileges system goes through the following steps:

1. It identifies the role encountering the check and gets all his ancestors.
2. For each ancestor it creates an irreducible set by finding all the assigned privileges and winnowing them.
3. Next, for each ancestor level the privileges inherited to the next level, where they are winnowed again, until the role is reached. The result of this process is an irreducible set of privileges specific to that role.
4. In a final step the privileges of the set are compared ome by one against the Schema of the security. If any of the privileges implies the Schema, the security check is passed.

In the following example:

```
            Everybody           offspring distance 2 or 3
            |
            |--Marketing        offspring distance 1
            |  |
            |  |--Product Mgr
            |
            |--Europe           offspring distance 2
               |
               |--Spain         offspring distance 1
                  |
                  |--Product Mgr
```

Let's assume the user logged in has the Product Mgr role and an action is requested by that user which is protected by the security system. Below are the steps what happens. Look at this example from the point of view of the actual role requesting the protected action (here: Product Mgr). The offspring distance is the ancestry relative to Product Mgr (the maximum value for it)

1. The system gets the privileges of all ancestors of Product Mgr and winnows the privileges of each of them. In this case: Everybody, Marketing, Europe and Spain. The privileges of the role itself are also fetched and winnowed. The result is that each node in the tree now has its irreducuble set of privileges assigned to it.
2. Next, each irreducible set is looked at, starting at the top level. (here: Everybody).
3. The irreducible set is inherited by the offspring at the next distance. In this case "Europe" inherits the irreducible set of "Everybody". (3->2 inheritance), and "Marketing" also inherits the irreducible set of "Everybody" (2->1 inheritance).
4. This process is repeated for the next distance level, so "Spain" inherits "Europe" (2->1)
5. The "Spain" and "Marketing" privileges are then winnowed against each other (both are at distance 1)
6. At this point we have the irreducible sets for distance 1, so we can continue to the requesting role. Product Mgr inherits the irreducable set from distance 1 (1->0) (which have been created from the irreducable sets from the higher distances). This produces the irreducible set of privileges for the Product Mgr role.
7. That set is compared against the mask of the security check needed for the requested action. If one of the privileges in the set implies the mask, the check is passed and the user is granted to perform the action.

You probably want to read the above again. ;-)

4. Architecture

4.1 Admin and User APIs

The following are the standard-type APIs used in Xaraya. The $args parameter represents an array of arguments to the function.

4.1.1 Admin API

- *roles_adminapi_create($args)*: create a new user.
- *roles_adminapi_delete($args)*: delete a user.
- *roles_adminapi_update($args)*: update a user.
- *roles_adminapi_stateupdate($args)*: update a user's state.
- *roles_adminapi_getmenulinks()*: passmenulinks to the main menu.
- *roles_adminapi_addgroup($args)*: create a new group.
- *roles_adminapi_getallgroups()*: generate a listing of all groups.
- *roles_adminapi_deletegroup($args)*: delete a group.
- *roles_adminapi_renamegroup($args)*: rename a group.
- *roles_adminapi_viewgroup($args)*: view the users in a group.
- *roles_adminapi_deleteuser($args)*: delete a user from a group.
- *roles_adminapi_addmember($args)*: add a user to a group.

4.1.2 User API

- *roles_userapi_getall($args)*: generate a listing of all the users..
- *roles_userapi_getallactive($args)*: generate a listing of all the active users.
- *roles_userapi_get($args)*: get a user.
- *roles_userapi_countitems()*: return the number of users.
- *roles_userapi_login($args)*: log a user in.
- *roles_userapi_makePass()*: generate a password.
- *roles_userapi_getmenulinks()*: return menulinks to the main menu.
- *roles_userapi_getallgroups()*: generate a listing of all the groups.
- *roles_userapi_getUsers($args)*: generate a listing of all the users in a given groups.
- *roles_userapi_countgroups()*: return the number of groups.
- *roles_userapi_addmember($args)*: add a user to a group.

4.2 Setup API

The following functions should be used to create default roles and privileges in the init.php of a module.

- *xarMakeGroup*: create a new group.
- *xarMakeUser*: create a new user.
- *xarMakeRoleRoot*: define an entry in the roles repository that is the head of a roles hierarchy.
- *xarMakeRoleMemberByName*: make one role the child of another.

- *xarMakeRoleMemberByUname*: make one role the child of another.
- *xarMakeRoleMemberByID*: make one role the child of another.
- *xarRegisterPrivilege*: create a privilege in the rpivileges repository.
- *xarMakePrivilegeRoot*: define an entry in the privileges repository that is the head of a privileges hierarchy.
- *xarAssignPrivilege*: assign a privilege to a role.
- *xarDefineInstance*: create an instance in the instance repository.
- *xarRemoveInstances*: remove all the instances of a module from the instances repository.
- *xarGetGroups*: returns an array representing all the groups.
- *xarFindRole*: finds a role by its name.
- *xarRegisterMask*: register a mask in the mask repository.
- *xarUnregisterMask*: remove a mask from the mask repository.
- *xarRemoveMasks*: remove all the masks of a module from the masks repository.
- *xarSecurityCheck*: check a the current user's privilege against a mask.

4.3 Classes and Methods

The classes and class methods should in general not be accessed directly. Most of them return objects rather than arrays, so use them at your own risk.

4.3.1 xarRoles

A central repository of user and group definitions. Each user, group, group of groups etc. is a role with an entry in this repository.

An implementation of the roles object conceptually replaces the tables xar_user and xar_groups. The code for manipulating roles will deal with the issue of whether the role is a group or a user and handle things accordingly.

Methods:

- *constructor*
- *getgroups*: returns an array representing all the groups in the system. Note: should be private.
- *getgroup*: returns a single role of type group based on its ID. Note: should be private.
- *getsubgroups*: returns all the subgroups of a given role based on its ID.
- *makeTree*: returns a tree representation of the roles of type group.
- *drawtree*: returns a crude HTML drawing of a tree generated by maketree. Note: needs to be moved out to the templates.
- *getRole*: returns a a role object based on its ID.
- *findRole*: returns a role object based on its name.
- *makeMemberByName*: makes one role a child of another. Uses the names as inputs.
- *isRoot*: creates an entry in the repository that is the head of the roles hierarchy.
- *makeUser*: creates a role object of type user in the repository.
- *makeGroup*: creates a role object of type group in the repository.

4.3.2 xarRole

An object representing a single group or user.

Methods:

- *constructor*
- *add*: add this role to the roles repository.
- *addMember*: add a child role to this role.
- *removeMember*: remove a child role from this role.
- *remove*: remove this role from the roles repository.
- *update*: update this role in the roles repository..
- *remove*: remove this role from the Roles repository.
- *getAllPrivileges*: get an array representing all the privileges defined.
- *getAssignedPrivileges*: get an array of privilege objects assigned to this role.
- *getInheritedPrivileges*: get an array of privilege objects assigned to this role's ancestors.
- *assignPrivilege*: assign a privilege to this role
- *removePrivilege*: remove a privilege from this role
- *getUsers*: get the children of this role that are users.
- *getParents*: gets an array of the role objects this role is a member of
- *getAncestors*: recursive getParents.
- *isEqual*: check whether this role is the same as another.
- *isUser*: check whether this role is a user
- *isParent*: check whether a role is a parent of this role
- *isAncestor*: check whether a role is an ancestor of this role
- *getPrivileges*: get an array of all the privilege objects defined.
- *gets* and *sets*: of all the relevant data items in the xarRole object.

4.3.3 xarMasks

A central repository of privilege masks (policies?). The masks are normally defined and registered as entries upon installation of a module. There needs to be a restriction that assigned names for components need to be unique within a module.

Methods:

- *constructor*
- *getmasks*: returns an array of mask objects for a given module and component.
- *register*: create an entry in the masks repository.
- *unregister*: remove an entry from the masks repository. (this method is deprecated)
- *removemasks*: removes all the masks of a given module from the masks repository.
- *winnow*: merges 2 arrays of mask objects (same level behavior).
- *trump*: merges 2 arrays of mask objects (inheritance behavior).
- *xarSecurityCheck*: checks a privilege against a mask.
- *getmask*: returns a single mask object based on its name.

4.3.4 xarPrivileges

A central repository for privileges defined at any given time. No duplicate privileges are allowed. This object

extends the xarMasks object.

Methods:

- *defineInstance*: creates an instance definition in the instance repository.
- *removeInstances*: removes all the instance definitions of a given module from the instance repository.
- *register*: create an entry in the privileges repository.
- *assign*: assign a privilege to a role.
- *getprivileges*: returns an array representing all the privileges in the system.
- *gettoplevelprivileges*: returns an array representing all the privileges in the system that heads of a compound privilege (this function is deprecated)..
- *getrealms*: returns an array representing all the realms in the system.
- *getmodules*: returns an array representing all the modules in the system.
- *getcomponents*: returns an array representing all the components of a module.
- *getinstances*: returns an array representing all the instances of a module and component.
- *getsubprivileges*: returns all the child privileges of a given privilege based on its ID.
- *getprivilegefast*: returns an array representing a single privilege based on its ID. Note: should be private.
- *tree functions*: a number of help functions for displaying compound privileges.
- *getPrivilege*: retrieves a privilege object from the privileges repository based on its ID.
- *findPrivilege*: retrieves a privilege object from the privileges repository based on its name.
- *makemember*: makes a privilege a component of another privilege.
- *makeentry*: defines a top-level entry in the privileges repository.

4.3.5 xarMask

An object representing a single mask.

Methods:

- *constructor*
- *implies*: compares two masks or privileges.
- *gets* and *sets*: of all the relevant data items in the xarMask object.

4.3.6 xarPrivilege

An object representing a single privilege. This object extends xarMask.

Methods:

- *constructor*
- *add*: add this privilege to the Privileges repository.
- *makeEntry*: adds this privilege as a toplevel entry to the privileges repository.
- *addMember*: adds a component to this privilege.
- *removeMember*: removes a component of this privilege.
- *update*: updates this privilege in the privileges repository.
- *remove*: removes this privilege from the privileges repository.
- *getRoles*: returns an array of roles objects this privilege is assigned to.
- *removeRole*: removes an ssignation of this privilege to a role.

- *getParents*: returns an array of privilege objects this privilege is a component of.
- *getAncestors*: recursive getParents.
- *getChildren*: returns an array of privilege objects that are components of this privilege.
- *getDescendants*: recursive getChildren.
- *isEqual*: compares this privilege object to another.
- *isEmpty*: returns true if this privilege is an empty container.

5. Database Tables

5.1 Roles Tables

The tables xar_groups and xar_users can be merged into a new Roles table. The table xar_group_membership also changes:

```
            Table xar_roles     Table xar_rolemembers
            xar_uid                  xar_uid
            xar_name                 xar_parentid
            xar_type
            xar_users
            xar_uname
            xar_email
            xar_pass
            xar_date_reg
            xar_valcode
            xar_state
            xar_auth_module
```

The field xar_type is 0 implies child is a user, 1 implies child is a group. Calls to groups and users in the API have to be appropriately modified to look up the new table. [MrB: evaluate whether we can use "implicit" typing. if(nochildren): user;else: group]

5.2 Masks Table

```
            Table xar_security_masks
            xar_sid
            xar_name
            xar_realm  [MrB: leave it for now, don't use it]
            xar_module
            xar_component
            xar_instance
            xar_level
            xar_description
```

5.3 Instances Table

```
            Table xar_security_instances
            xar_iid
            xar_module
            xar_component
            xar_header
            xar_query
            xar_limit
            xar_description
```

5.4 Privileges Tables

```
            Table xar_privileges      Table xar_permmembers
            xar_pid                   xar_pid
            xar_name                  xar_parentid
            xar_realm
            xar_module
            xar_component
            xar_instance
            xar_level
            xar_description
```

5.5 ACL Table

```
            Table xar_security_acl
            xar_partid
            xar_privid
```

6. Syntax

6.1 Registering Masks

A mask is a special kind of privilege used to check other privileges. When a mask is encountered it is checked against the user's privileges to see whether access to the resource in question is granted. The resource a mask protects is called a component.

Each mask that will be used for security checks needs to be registered. This is done with the function xarRegisterMask, which has the following syntax:

```
        function
xarRegisterMask($name,$realm,$module,$component,$instance,$level,
        $description='')

        where:

        $name        : a name given to the mask. The name needs to be unique
within the
                       module.
        $realm       : the realm the mask applies to.
        $module      : the name of the module the mask applies to.
        $component   : the name of the component the mask belongs to. The
component name is
                       referenced during security checks.
        $instance    : the name of the instance the mask applies to. This refers
to an
                       instance that has been defined at runtime (see below),
        $level       : the security level a privilege must have to pass the
check. These
                       are the usual values 0 - 800.
        $description : a text field that describes the mask.
```

All fields except the description are mandatory. The value 'All' can be used for realm, module, component, instancetype or instance. In the case of instances, expanding 'All' to the number of instance types makes your definition clearer, as shown in the example below:

```
xarRegisterMask('EditClassification','All','articles','Classiciation','All:All:All',ACCESS_EDIT)
```

This creates a mask named EditClassification for the articles module. Security checks can invoke the mask as described below.

Masks need to be registered at init time. In other words, for each mask to be registered, invoke the xarRegisterMask function in the init.php of the module the mask belongs to.

Such a call could also be inserted when a new instance is created. This would let have the user define masks at run time. (can of worms, here) [MrB: yeah, don't allow it initially till we have worked it out how exactly to do that]

6.2 Defining Instances

Security checks in a module need to check against specific instances. We therefore want to ensure that the instances administrators include in the privileges they create are well formed. This is done by registering the instance definitions with the system,

Instance definitions are not strictly necessary to make the security system work, but they help make creating privileges easier.

Instances conceptually are objects that a module deals with, e.g. articles, folders, download items, dynamic data fields. They can be created initially or at run time.

An instance can usually be defined by one or more database fields. For example an instance in the categories module, a category, can be defined by its title, or even more precisely by its ID, which is a unique reference.

Module developers will use several such "filters" to define instances. Postnuke limited their number to 3, but Xaraya allows any number.

Instances are defined by the xarDefineInstance function:

```
        function xarDefineInstance($module, $component, $instancedefinition)

        where

        $module             : module to which the instance applies.
        $component          : component which the instance is part of.
        $instancedefinition : an arry that represents the definition of the
instance.
                              The array contains n entries 3 elements each:
                               - a header text
                               - a field definition given by an sql query
                               - a parameter giving the maximum number of instance
items
                                 to be shown.
```

Each instance to be defined must invoke this function in the init.php file of the module it refers to.

Each entry in the instance definition array defines one "filter" for the instance (such as a category title). The elements in the array entry let the UI create a dropdown that is used when creating privileges. The header text can be displayed to indicate the type of "filter". The sql statement defines what database entries will be shown in the dropdown. The limit parameter can be set by the module developer to limit the number of dropdown items. If the actual number of instances is greater than the limit, the UI will show an empty text field for manual entry instead of a dropdown.

An example of how this might be used:

```
        $query1 = "SELECT DISTINCT xar_pubtypeid FROM xar_articles";
        $query2 = "SELECT DISTINCT xar_cid FROM xar_categories";
        $query3 = "SELECT DISTINCT xar_authorid FROM xar_articles";

        $instances = array(
                        array('header' => 'Pub. Type ID:',
                              'query' => $query1,
                              'limit' => 20
                             ),
                        array('header' => 'Category ID:',
                              'query' => $query2,
                              'limit' => 20
                             ),
                        array('header' => 'Author ID:',
                              'query' => $query3,
                              'limit' => 20
                             )
                    );
```

With this definition (which is created in the init.php of the articles module) the UI will create 3 dropdowns for defining instances that are article, namely publication type, category and author, all given by their respective IDs. The option 'All' is automatically added to each dropdown.

If more than, for instance, 20 categories have been defined, the UI will substitute a text box for the second dropdown, requiring the administrator to manually enter a category ID when defining a privilege. Note that no validity checks are performed on such input.

Note that, although the UI can ensure that the ID choices for publication type, category and author are valid, there is no guarantee that an article corresponding to a given combination exists or will exist. In other words, with instance definitions the UI can ensure that privileges contain well formed instances, but it *cannot* ensure that the instances are valid.

6.3 Security Checks

Security checks are the way the system checks a user's a privileges against a mask of a component. Security checks replace the previous xarSecAuthAction function. A simple example of a security check is:

```
        if(!xarSecurityCheck('EditArticles')) return;
```

On encountering this line in the code, the system will check the user's privileges against those of the mask 'EditArticles'. If the mask is implied by at least one of the user's privileges, the function returns true. If the security check fails the function returns false. Like the xarSecAuthAction function, securitycheck will also display a standard error message if the check fails.

The error message can also be suppressed, as shown in the following example:

```
        if (xarSecurityCheck('EditArticles',0)) {

           ...code to edit an article in the articles module

        }
```

Here the user's privileges are again checked against 'EditArticles' to see whether the user may add an article. The 0 value tells the function not to display an error message if the check fails.

The full syntax of the xarSecurityCheck function is:

```
         function
xarSecurityCheck($name,$catch=1,$component='',$instance='',$module='',$role='')

         where

         $name         : name of the  mask to be checked.
         $catch        : 1 = show the exception message if the check fails.
                         0 = do not show an exception message
         $component    : name of a component to be checked against.
         $instance     : name of an instance to be checked against.
         $role         : name of a role to be checked against.
         $module       : name of a modules to be checked against.
```

The component, instance, role and module values are used to override values defined in the mask's definition. For instance, entering a value for $component and $instance lets you check against an instance defined at runtime, rather than when the mask is registered.

The default values for $role and $module are the current user and the module in which the security check is encountered. Entering a value for $role lets you check against the privileges of a user other than the current user.

7. Converting Modules from the Old to the New System

Here is a step by step guide to switching modules from Postnuke's (or Xaraya's previous) permissions system to the Xaraya security system.

1. Create the necessary instance definitions
2. Create the necessary mask definitions
3. Change the pnSecAuthAction or xarSecAuthAction calls to xarSecurityCheck calls
4. Remove redundant code

7.1 Creating Instance definitions

Defining instances is a bit of work. The good news is that there are probably not more than 3 of these in any given module.

The first question to answer is: what instances do I have or need? Instance definitions ("security schemas") can generally be found in the version.php file of the module. However, you'll probably also want to look through the module code for calls of the pnSecAddSchema or xarSecAddSchema function, which also define security schemas. You'll need to create an instance definition for each of the entries in a security schema.

Security schemas have the form:

```
        $modversion['securityschema'] = array('categories::category' =>
                                              'Category name::Category ID',
                                              'categories::item' =>
                                              'Category ID:Module ID:Item ID');
```

In this case we have a schema with two entries, which will call for two instance definitions. In the first entry the line

```
        'categories::category'
```

refers to the module name ("categories") and the component name ("category"). Both of these can be inserted into the instance definition as shown below. If there is no component name, insert "All" in the definition instead.

The second part of the entry is a bit more complicated:

```
        'Category name::Category ID'
```

This part of the schema consists of up to 3 names of database fields which make up the instance definition. You need to define the sql query strings which return a recordset with each of these fields from the database. In this case the query strings would be:

```
        $query1 = "SELECT DISTINCT xar_name FROM xar_categories";
        $query2 = "SELECT DISTINCT xar_cid FROM xar_categories";
```

Note you'll have to look in the database a bit to identify the correct table and field names. Unfortunately this is not clearly documented anywhere, but in most cases the table will be one of those belonging to the module itself, and the field names can be identified fairly easily.

Once you have the module, component names and the query strings these can be inserted in the instance definition. The definition for the first entry in the security schema above is shown below, with appropriate values for the header and limit parameters included:

```
        $query1 = "SELECT DISTINCT xar_name FROM xar_categories";
        $query2 = "SELECT DISTINCT xar_cid FROM xar_categories";
        $instances = array(
                        array('header' => 'Category Name:',
                              'query' =>  $query1,
                              'limit' =>  20
                             ),
                        array('header' => 'Category ID:',
                              'query' =>  $query2,
                              'limit' =>  20
                             )
```

```
                            );
        xarDefineInstance('categories','Category',$instances);
```

7.2 Creating Mask Definitions

The next step is to create masks for all the security checks in the code. These are privilege definitions which are referenced whenever a security check is called: the check compares the user's privileges with the privilege defined by the mask and decides whether the user passes the check or not.

There are typically 3-6 different masks for any given module. How do I know what masks to define? Unfortunately the only way to figure that out is to go through the code searching for pnSecAuthAction or xarSecAuthAction calls and seeing how many different types there are.

Here is a useful rule: Assume 1 mask to be created for each security level you find in the *SecAuthAction calls.

Once you have identified the different calls, create a mask definition for each of them in the init.php of the module. Use the following function:

```
        Syntax:
xarRegisterMask(Name,Realm,Module,Component,Instance,Level,Description)

        Example:
xarRegisterMask('ReadCategoryBlock','All','categories','Block','All:All:All',ACCESS_READ);
```

- "ReadCategoryBlock" is the name we give the mask. This is referenced in the code when performing security checks. The name must be unique within the module the mask applies to, in this case categories. It is also wise to choose a name that describes the rights the mask contains.
- The realm parameter indicates which realm the mask belongs to.
- "Categories", as noted, is the name of the module the mask belongs to. Module names by convention are spelled in lowercase.
- "Block" is the name of the component the mask belongs to. Component names by convention are spelled in uppercase.
- The instance parameter in general is left at 'All' and overriden at run time. In this example 'All:All:All' indicates that this component supports 3 instance "filters". ('All' would also be valid)
- The level parameter indicates the kind of access required for the check to be successful.
- The Description parameter is optional.

7.3 Changing the Security Check Calls

The next step is the most labor intensive: changing the *SecAuthAction calls to the new system using xarSecurityCheck. A module may contain literally dozens of these calls sprinkled through the code. Unfortunately each needs to be visually examined and appropriately modified.

Some examples are shown below. In each of these we assume an appropriate mask has been created based on the information in the old call, and that can be referenced by the new call.

7.3.1 Security Check with Exception Catching

```
        if (!xarSecAuthAction('All', 'categories::', "::", ACCESS_EDIT)) {
return;}

        becomes

        if (xarSecurityCheck('EditCategories')) return;
```

The xarSecuritCheck call references the mask we defined above via the mask name. The module name and access level information is stored in the mask, so there is no need for them as parameters in the call. If the

security chekc fails the system will automatically generate an exception message.

7.3.2 Security Check with Exception Catching Suppressed

```
        if (xarSecAuthAction('All', 'categories::', "::", ACCESS_EDIT)) {

        becomes

        if (xarSecurityCheck('EditCategories',0)) {
```

This is identical to the previous call except that the exception message that would be generated in the case of a failed security check is suppressed by setting the second parameter to 0.

This call is useful when you want to loop through a series of security checks in the code without halting execution if the check fails.

7.3.3 Security Check with Dynamic Components or Instances

```
        if (!xarSecAuthAction(0, "categories::item", "$name::$id", ACCESS_EDIT))
           { return; }

        becomes

        if (!xarSecurityCheck('EditCategories',0,'Item','$name:All:$id')) return;
```

In this example we override some of the mask definition parameters with dynamic values. Recall that the mask definition above is:

```
        Syntax:
xarRegisterMask(Name,Realm,Module,Component,Instance,Level,Description)

        Example:
xarRegisterMask('EditCategories','All','categories','All','All',ACCESS_EDIT);
```

In this definition both the Component and Instance parameters are "All". By adding the parameters "item" and "$name::$id" to the xarSecurityCheck call we replace the values "All" by dynamic values to be checked against.

Note that "$name::$id" in the old call becomes "$name:All:$id" in the new call. We explicitly list all of the instances "filters", even those that in the old call are not mentioned.

It is important for there to be consistency among instances, masks and security checks. If a security check does not find a mask of the name called, it will throw an exception. If the check includes dynamic component and/or instance values that do not coincide with an instance definition, it may fail.

7.3.4 Removing Redundant Code

The following code becomes redundant after the conversion and can be removed:

1. The old *SecAuthAction calls: can be removed immediately after being replaced by xarSecurityCheck. These calls need to be removed for the system to work.
2. Any calls to *SecAddSchema in the code: the calls do nothing; if left the new system still works.

We recommend leaving the security schema definitions in the version.php files for documentation purposes. However, they should accurately reflect the instances used in the xarSecurityCheck calls.

8. Open Issues

8.1 Regex

I'd rather dispense with this and run the UI through dropdowns and checkboxes, but I understand that some people are clamoring for an "advanced user" UI for the privileges system. There may also be some performance considerations here, trading CPU processing for DB hits. And then there's backwards compatibility. Well, OK.

My suggestion for the future would be to limit the advanced UI to entering lists into the appropriate form fields, say a "Component" or "Instance" field in the UI. On submitting, the list(s) would be disassembled to records in the privileges table with one component/instance per record. This in itself is not trivial, because the user would be entering some sort of name, and you'd have to parse the name, make sure it corrsponded to a real object registered in the privileges system, and then translate it to a numeric ID.

8.2 Multi-language

Need to check to make sure any calls using the name of an object rather than its ID don't screw up the ML capabilities.

8.3 Realms

Realms have been in the old security system, but no-one seems to know exactly what they do and how to use them. Until that is crystal clear, realms will not be supported, or rather, actively not supported. We can't have loose ends in a security system

8.4 Future Implementation

The issues encountered in Version 1.0 suggest merging the Roles and Privileges modules into a single module in the future.

A thourough evaluation needs to be done, whether we want the security system in user module space. This has both advantages and disadvantages. For ease of migration they will be in user module space for now, possibly classified as "Core Admin", so a reasonable guarantee can be given they exist when bootstrapping the system.

8.5 Miscellaneous Notes To Do

- Need to integrate a regex translation layer for backward compatibility.
- We need to set up a quality assurance program in general for Xaraya, but specifically for security related issues. If we ever want Xaraya to be used in corporate environments, this is a critical requirement.

9. Revision history

Version 0.9, Dec 15, 2002: First release of this document

Version 1.0, Jan 15, 2003: Major rewrite based on the first version of the code. Clearer definition of the architecture, implementation and UI.

Version 1.1, Feb. 8, 200: terminology changes, review by MrB

10 Reference title

Author's Address

Marc Lutolf
Xaraya Development Group
EMail: marcinmilan@xaraya.com
URI: http://www.xaraya.com

Intellectual Property Statement

The DDF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the DDF's procedures with respect to rights in standards-track and standards-related documentation can be found in RFC-0.

The DDF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the DDF Board of Directors.

Acknowledgement

Funding for the RFC Editor function is provided by the DDF