OFBiz Security Redesign

This page is intended to be used as a means of collaboration on the Apache OFBiz security redesign.

A new design is being proposed, but it is not finalized. Anyone who is interested is welcome to comment, critique, and add suggestions.

Security-Aware Artifacts

Introduction

The current OFBiz security implementation uses indirect security control - where permission services are used to control access to OFBiz artifacts. Permission services are small scripts that evaluate user permissions, determine if a user is related to a piece in data in some way, and other tasks. In the end, the script returns an access-granted or access-denied result (hasPermission) that is used to control access to the artifact(s) managed by the script.

As the number of artifacts in OFBiz grows, the number and complexity of permission services grows too. Some of this growth can be controlled by having common permission checking done in shared scripts.

Permission services use a coarse set of permissions - providing a basic set of create/update/delete (CRUD) flags that control broad ranges of artifacts. Fine-grained control is possible, but it requires complicated conditional processing.

A permission service can be extended by assigning a Service Event Condition Action (SECA) to it. If the primary permission service returns hasPermission=false, the SECA is triggered and an additional permission service is run. The result of the second permission service controls access to the artifact(s).

Disadvantages

Since the current OFBiz security implementation uses indirect security control, there is no direct connection of permission logic to the artifact that the logic is trying to control access to. Access to a particular artifact could be (and often is) controlled by a handful of different scripts. While this may appear to be an advantage (because it's flexible), it becomes a problem when something about the artifact changes that requires a change in access control. Permission scripts that control the artifact must be tracked down and updated.

Permission checking is optional. Through oversight or laziness artifacts can be left unprotected - opening up security holes or attack vectors.

The coarse grained permission logic fosters an "all or nothing" design mentality. Smaller artifacts - like menu items and data entry fields - can be controlled individually by permissions, but it requires cumbersome conditional code. Many times the developer will just give the user access to all of these smaller artifacts - regardless of whether or not the user has permission to use them.

The only way to get a list of all the user's permissions associated with an artifact is to perform a permission check for every one of the CRUD permissions and add each result to a list. That makes artifact access control cumbersome.

Any time a new artifact is created that needs access control, a new script needs to be written to control it.

A Different Approach

What if we took the job of artifact access control away from the permission services and gave it to the artifacts themselves? What if every artifact was security-aware?

Instead of an artifact's access being controlled in a handful of places, it is controlled in one place - within the artifact itself. There is no complicated intermediary code that must consider all possible interactions a user might have with a set of artifacts. Since all artifacts are security-aware, fine grained control is possible. There is no need for cumbersome conditional logic to control an artifact - the artifact controls itself.

Design Overview

There are three key players in the Security-Aware Artifact design:

1. The user.
2. The OFBiz artifact - an application, a service, an entity, or one of the screen widgets.
3. An authorization manager (permissions management software).

The process is very simple:

1. The user is about to interact with an artifact (through an event or a request).
2. The artifact takes the UserLogin GenericValue and the artifact's identifier, and hands them to the authorization manager. The authorization manager returns a list of permissions granted to the user for that artifact.
3. The artifact performs some type of default behavior based on the list of permissions returned.

Let's take a closer look at the process.

A user action triggers an interaction with an artifact - a service is invoked, or an entity is queried for data, or a screen widget is about to be rendered. In all cases, the security-aware artifacts (the service, the entity, or the screen) are responsible for their own security-related behavior.

Before the artifact begins its intended task, it asks the authorization manager for the permissions the user has for the artifact. In effect, the artifact says to the authorization manager, "Give me user X's permissions for artifact Y."

The authorization manager returns a list of permissions. For the sake of illustration, let's say the mere presence of a permission grants that permission. So, if "create" is in the permission list, then the user has create permission for the artifact.

What happens next depends upon the type of artifact - that is called the default behavior.

If the artifact is an entity, then the permissions control the user's ability to view, create, update, and delete records. If the artifact is a screen widget, then the permissions control if and how things are displayed. If the artifact is a service, then the permissions control if the service can be invoked.

Default behaviors for types of artifacts will be well documented, so the developer and OFBiz administrator will know exactly how permissions will affect each artifact.

Design Details

If each artifact has its own set of permissions, there could be potentially thousands of permissions - an administration nightmare! It would be helpful to have artifact permissions organized in a hierarchy - so that permissions defined higher up in the hierarchy are inherited by artifacts lower in the hierarchy. The hierarchy will reduce the number of permissions needed.

An execution path could be considered a hierarchy. We can build on that structure to set up a hierarchy of security-aware artifacts:

- OFBiz
  - Component
    - WebApp
      - ControllerRequest
        - ControllerRequestEvent
          - Service
            - Entity
              - EntityField
      - ControllerView
        - WidgetScreen
          - Service
            - Entity
              - EntityField
          - FtlFile
            - WidgetScreen
          - WidgetForm
            - WidgetFormField
  - ...

Permissions assigned at the OFBiz/component/webapp level are inherited by all artifacts below that level. Permissions assigned at lower levels replace any inherited permissions. Permission inheritance is managed by the authorization manager.

In the design overview, it was mentioned that the security-aware artifact provides its identifier to the authorization manager when requesting permissions for itself. That identifier must be unique in order for the system to work. We can leverage the artifact hierarchy to construct a unique artifact identifier. The authorization manager should handle the identifier in a case-insensitive way.

The "OFBiz" portion of the identifier is considered the root of the hierarchy.

The permission list returned by the authorization manager is a set of flags - implemented as key/value pairs. The flag key indicates the type of permission - view, create, update, delete, etc. Valid flag values are "true" or "false." A value of "true" indicates access-granted, and a value of "false" or the absence of a key/value pair indicates access-denied. The permission list could consist of the standard view, create, update, and delete permissions. Artifacts are allowed to define additional permissions. For example, a web application or service artifact might need only one permission - access.

It is common for services to invoke other services, and for entities to be related to other entities. In those cases, the secondary or "child" artifacts inherit the permissions of the initial service or entity artifact. Permissions assigned to the child artifacts replace any inherited permissions.

Some of the existing permissions service scripts go beyond a simple check if a permission exists - they also check to see if the user is related to the data being accessed or changed. In other words, user access to data is constrained in some way depending upon the user. In those cases, we need more than a simple access-granted/access-denied permission - we need a service to perform those additional tests and we need a way for the authorization manager to specify the service.

When an artifact requires additional permissions checking beyond a simple list of permission flags, the additional permission checking logic will be put in a service, and the service will return an access-granted/access-denied result (just like the existing permission services). When that artifact requests permissions from the authorization manager, the authorization manager returns the normal list of permission key/value pairs plus a list of permission-checking service key/value pairs - where the key is "service" and the value is the service name. The permission service list is a set (no duplicates) and it represents an intersection - all services in the list must return an access-granted condition for access to be granted to the artifact.

When an artifact receives a list of permission services in addition to the list of permission flags, it will process the permission flags first, then the list of permission services.

The existing permission services can be reused in the new design. Many of the existing permission services check a permission flag (mainAction), and then do some additional processing. To convert those permission services over to the new design, just remove the permission flag check.

A special case exists for entity lists. Sometimes it is desirable to filter a list of records based upon the user. Treating each record as an artifact and performing permission checks on them one at a time would be inefficient. We need a way to specify a filter.

The entity artifact could support permission "filters" - where a script is run to process a list of records. In those cases, the authorization manager would include the filters in the permission list - where the key is "filter" and the value is the script or service name.

There are two ways the filter could be implemented:

1. The entity engine passes an unfiltered list of entities to the service, the service picks matching records from the list and returns them. The advantage to this approach is we can use existing code to implement it. The disadvantage is it is inefficient.
2. The service returns an EntityCondition that represents the constraint, and the entity engine merges that EntityCondition with any condition it already has. The advantage to this approach is the list of entities is filtered by the database. The disadvantage is there is no way to return an EntityCondition in mini-language. Mini-language would have to be extended, or the EntityCondition would have to be constructed in a Beanshell or Groovy script, or the filters would have to be written in Java.

The Authorization Manager

The Authorization Manager (AM) is a piece of software responsible for managing users and and their permissions.

The AM supports grouping users into User Groups. The OFBiz administrator will create Groups appropriate for his/her organization, and then assign users as members of those groups.

Groups can be members of each other. For example, user X can be a member of User Group Y, and User Group Y can be a member of User Group Z. Users and groups can be members of more than one group.

The AM supports assigning artifact permissions to users and User Groups. Assigning artifact permissions to a user or group simply means a connection is made between the user or group and a set of artifact permissions. In other words, the user or group "points to" a set of artifact permissions.

When permissions are assigned to a group, all members of that group share those permissions. This gives the OFBiz administrator tremendous flexibility in assigning permissions. Permissions can be assigned directly to a user, or they can be assigned to a User Group - and then the user is made a member of that group.

When a user or group is a member of more than one group, the permissions from all groups are combined in a set.

Another way to view the AM user/user group design is by using properties:

Authorization Manager Artifact	Properties
User	IsUser, HasPermissions, IsMember
User Group	HasPermissions, IsMember, HasMembers

The Authorization Manager must support the following operations:

1. Create/update/delete users.
2. Create/update/delete User Groups.
3. Create/delete group memberships.
4. Create/delete artifact permission assignments to users and groups.
5. Return a list of permission flags and permission services for a given user/artifact pair.

Other operations could be included to support a security administration user interface.

The AM is a security-aware artifact. Users must have the appropriate permissions to perform any of the AM create/delete operations.

The AM's task of returning permission flags and permission services has been referred to throughout this document. Let's summarize that process from the AM's perspective.

An artifact provides a UserLogin GenericValue and an artifact identifier to the AM for a permissions request. The AM uses the UserLogin to look up the user's permissions in external storage. This would include all groups the user is a member of, and the groups that those groups are members of, etc. The AM compiles a list of the user's permissions, searches the list for the artifact's identifier, and then returns whatever permissions are found.

For the sake of illustration let's say the AM builds an internal tree structure from all of the user's permissions. The tree structure would look just like the artifact hierarchy - but it only includes artifact identifiers the user has permissions for. Using the requesting artifact's identifier (and working from left to right), the AM "walks the tree" - looking for permission flags and permission services. While the AM walks the tree, it keeps a copy of the last permissions found. Every time new permissions are found, they replace the previous copy (that is how permission inheritance is achieved). Eventually, the AM will encounter the end of the tree or the end of the artifact identifier. When that happens, the AM returns the last permissions found.

The OFBiz User Group

Sometimes it is desirable to assign some basic permissions to ALL OFBiz users. Mailing list comments indicate that some installations meet that need by using a custom CreateUser service, or a SECA that adds the permissions to new users, or some other solution.

It would be preferable if OFBiz could accommodate permission assignments for all users.

In this design, that need is fulfilled by having a default (or "out-of-the-box") User Group called OFBiz Users. The Authorization Manager will make all new users a member of that group, and consequently that group will be included in all permission checks. The Authorization Manager will throw an exception any time an attempt is made to delete the "OFBiz Users" user group.

The Not-Logged-In User

The Security-Aware Artifact design introduces a fundamental problem: How do you get a user logged in? The various artifacts needed to log in a user won't operate without a UserLogin GenericValue present (and necessary permissions assigned to that user).

In this design, the problem is solved with a Not-Logged-In user account. That user's account is disabled, and it is assigned only the permissions needed to get a user logged in. Front-end processors (event/request handlers, etc) create the Not-Logged-In UserLogin GenericValue so that artifacts needed for login can be used.

The Super User

It is common to have a way to identify an administrator user, or super user. There are two ways that can be accomplished in this design:

1. Have an admin permission.
2. Have an admin User Role.

Although the term "admin" implies a user role, it would be more flexible to make it a permission. A user could be granted the admin permission to any artifact, which would make the user the admin of that artifact and all the artifacts below it. If an "Admin" User Role was desired, the admin permission could be assigned to a User Group called "Admin User Role" - and all the members of that group would become admins.

A user could be made an admin of all of OFBiz by granting the user admin permission to the OFBiz (root) artifact.

Artifacts must treat the admin permission as expected - it is the same as having all permissions. If the admin permission appears in a list of permission flags, it takes precedence - all other permission flags and permission services are ignored.

Artifact Reuse

Handling permissions with reused artifacts is greatly simplified with the Security-Aware Artifacts design. The problems associated with hard-coded permissions are eliminated.

When a screen widget artifact is reused in another component, that artifact's identifier changes - since its execution path is different. For example, if I reuse the Example component's EditExample screen in my specialpurpose/NewApplication component (by including it in my own screen definition), the reused EditExample screen artifact identifier changes from OFBiz/example/screen/EditExample to OFBiz/NewApplication/screen/EditExample. Permissions assigned to the latter will not grant access to the former.

The form in the reused EditExample screen will generate an updateExample event which, in turn, invokes the updateExample service. Just like the EditExample screen, the updateExample service artifact's identifier changes - I can give the users of NewApplication permission to access OFBiz /NewApplication/updateExample/updateExample without giving them access to any other Example component artifacts.

Transition

The existing security design and the new Security-Aware Artifact design can coexist. This ability will ease the transition to the new design.

Initially, the OFBiz User Group will have admin permissions assigned to it - one for each component (or group of artifacts). This will effectively disable the security checking in the new design - since the admin permission grants access to everything. The existing hard-coded permission checks will take care of security. As each component is converted over to the new design (hard-coded security checks are removed), the OFBiz User admin permission for that component is removed and the new design takes over security checking for that component (or group of artifacts).

Proposed Implementation Details

DEJ20090528: After thinking about this data model more and discussing it with a couple of others I don't think we should use what I have outlined below and instead we should create a more normalized data model. The main difference would be a few different entities in place of the bloated and denormalized SecurityArtifactAuth entity.

Existing Entities to Use:

UserLogin
UserLoginSecurityGroup
SecurityGroup

New Entities to Add:

SecurityArtifactAuth
-securityArtifactAuthId*
-securityGroupId
-securityArtifactGroupId
-authTypeEnumId (NotSpecified, Allow, Deny, AlwaysAllow (overrides Deny))
-authorizedActionEnumId (View, Create, Updated, Delete, All)
-recordViewEntityName
-recordFilterByDateEnumId (True, False, ByName)
-userMatchFieldName - for record level matching will lookup in recordViewEntityName view-entity for the currently logged in userLoginId or partyId, and if this field is set it will use this field name to match against, otherwise it will look for the userLoginId field or the partyId field in the view-entity
-authServiceName - this service will be called according to the pattern of current auth services and must return true/approved in addition to anything else passing that is populated in this entity

SecurityArtifactAuthConstraint
-securityArtifactAuthId*
-securityArtifactAuthConstraintSeqId*
-fieldName
-operatorEnumId (less-than, greater-than, etc)
-value (use ${} to expand from variable in artifact's context)

SecurityArtifactGroup
-securityArtifactGroupId*

SecurityArtifactGroupMember
-securityArtifactGroupId*
-artifactName* (can be securityArtifactGroupId; includes location#name when applicable)
-artifactTypeEnumId (Component, WebApp, ControllerRequest, ControllerRequestEvent, ControllerView, WidgetScreen, WidgetForm, WidgetFormField, FtlFile, Service, Entity, EntityField, ...)
-inheritParentArtifactAuth (Y, N) - defaults to Y; if Y then the user will have authorization for anything called by the artifact; if N user will need to have authorization for anything else called by the artifact

NOTE: we may want to move some fields between SecurityArtifactAuth and SecurityArtifactGroupMember as some things could go in either place, but should only go into one and which one they make sense in is a good question (and I've taken a pass here, but we may want to change it)

Possible Entities to Add:

-Could use these to make view-entity configurations on the fly (this may be WAY too technical for most users, so considering this a pretty low priority)

DynamicViewEntity
dynamicViewEntityId*

DynamicViewEntityMember
dynamicViewEntityId*
entityAlias*
entityName

DynamicViewEntityAlias
dynamicViewEntityId*
entityAlias*
fieldAlias*
fieldName
functionEnumId

DynamicViewEntityJoin
entityAlias
relEntityAlias
keyName
relKeyName
relOptional

Order of Artifact Checking - For WebApp Request

An ExecutionContext object is maintained that has an artifactAccessStack. In the artifactAccessStack as each artifact calls/users another artifact it is pushed onto the stack, and when it finished it pops itself off the stack. With this at any point we can see which artifacts were used to get to where we are. For the higher-level webapp-specific artifacts this will be kept in the request, for the widgets this will be kept in their context, for services it will be in the context, and for entities it will be in a thread-local variable. This is called an ExecutionContext because it may be used for more than just security /authorization purposes, and in fact we should refactor the current code that keeps track of the user and other things in ThreadLocal variables (mostly for the entity engine) so that they are in this ExecutionContext object.

The result of each artifact access control check will include:

- authTypeEnumId (NotSpecified, Allow, Deny, AlwaysAllow (overrides Deny))
- inheritSubArtifactAuth (Y, N)

Whenever an artifact is hit the authorization (access control) will be checked and the results of it will be kept in the artifactAccessStack. In order to support the inheritParentArtifactAuth field, as part of the ExecutionContext we will maintain a parentAuthorizationState variable that keeps track of the authTypeEnumId of the last artifact that had inheritParentArtifactAuth=Y. If an artifact called by the parent artifact passed but does not have inheritParentArtifactAuth=Y then the parentAuthorizationState variable will not be changed.

David, I don't think the ExecutionContext object should be concerned with security or the permissions hierarchy - that is the Authorization Manager's responsibility. All the ExecutionContext needs to provide is an artifact ID that the artifact can pass on to the Authorization Manager. - Adrian

The reason for this is that depending on the parentAuthorizationState the permission still needs to be checked but the results are interpreted differently:

- if parentAuthorizationState=NotSpecified:
  - authTypeEnumId interpreted as-is
- if parentAuthorizationState=Allow:
  - authTypeEnumId=NotSpecified: auth passed
  - authTypeEnumId=Allow: auth passed
  - authTypeEnumId=Deny: auth failed
  - authTypeEnumId=AlwaysAllow: auth passed
- if parentAuthorizationState=AlwaysAllow:
  - authTypeEnumId=NotSpecified: auth passed
  - authTypeEnumId=Allow: auth passed
  - authTypeEnumId=Deny: auth passed
  - authTypeEnumId=AlwaysAllow: auth passed

Note that parentAuthorizationState will never be set to "Deny" because before that would happen the auth would have failed and an error sent back up the stack.

Execution Context Implementation

Details can be found here.

Authorization Manager Implementation

The existing security classes will be refactored so the Security abstract class is converted to an interface, and the OFBizSecurity class will implement the interface. The Authorization Manager methods will be added to the Security interface, and the existing methods will be deprecated. This will make the conversion backward-compatible with existing installations. Details

The security classes have been refactored and committed. The Authorization Manager methods will be a separate interface, and the Security interface will extend it.

Access Control Scenarios

1. User X can use Artifact Y for anything that artifact supports and on any data (where "artifact" is a screen, web page, part of a screen or page, service, general logic, etc). Details

2. User X can use Artifact Y only for records determined by Constraint Z. Details

3. User X can use any artifact for records determined by Constraint Z. Details

4. Artifact Y can be used by any user for any purpose it supports. Details

5. Artifact Y can be used by any user only for records determined by Constraint Z. Details

6. User X can use any artifact for any record (ie superuser). Details

NOTE DEJ 20090514: the Proposed Implementation Details above currently does not support #3, but supports all others pretty well.