Implementación de un modelo "Role Based Access Control" en un entorno productivo

Juan Carlos Ramos[1] y María Luciana Roldán[1,2]

[1] Universidad Tecnológica Nacional – Facultad Regional Santa Fe, Santa Fe, Argentina
[2] INGAR, Instituto de Desarrollo y Diseño (CONICET/UTN)
jramos@frsf.utn.edu.ar, lroldan@santafe-conicet.gov.ar

Abstract. La complejidad de la administración de la seguridad en la gestión de múltiples sistemas de información es uno de los problemas más desafiantes a los que se enfrentan los desarrolladores y responsables de la ciberseguridad. El Control de Acceso Basado en Roles (RBAC) se ha convertido en el modelo predominante para el control de acceso avanzado porque disminuye el esfuerzo relacionado con el manejo de los permisos en un sistema. El valor de negocio de RBAC resulta de integrar seguridad y políticas de acceso con una vista organizacional de requerimientos de accesos. Esta integración es habilitada por tecnología, pero es además es una iniciativa que permite alinear políticas, procesos y responsabilidades organizacionales. Actualmente, las organizaciones cuentan con más de un sistema de información implementados en tecnologías diversas y que se encuentran desplegados en diferentes servidores, sobre los cuales se necesita implementar políticas de control de acceso en diferentes niveles. Sin embargo, las soluciones de control de acceso única u "off-the-shelf" no siempre aplican en forma directa, por lo que se requiere de soluciones a la medida de las necesidades y características de las organizaciones. En este trabajo se propone un modelo conceptual para control de acceso basado en roles basado en el estándar RBAC, y se define un esquema de implementación de permisos para el manejo de usuarios y roles en una organización, el cual se implementa en un entorno productivo.

Keywords: Control de acceso, Roles, Políticas de acceso.

1 Introducción

Los sistemas de control de acceso actúan como una primera línea de defensa contra la entrada no autorizada. Un mecanismo de autorización a los usuarios (asignación de permisos y privilegios) permite garantizar que los usuarios tengan los permisos correctos para realizar las actividades que le corresponden según su rol en el sistema.

Existen diversos estándares o modelos que establecen los fundamentos para el control de acceso [1] como el Control de Acceso Basado en Roles (RBAC, por su sigla en inglés) y el Control de Acceso Basado en Atributos (ABAC, por su sigla en inglés) [2]. Existen otros modelos de control de acceso como el Control de Acceso Discrecional (Discretionary Access Control) [3] en el que se incluyen mecanismos como 'Listas de Acceso' (Capability/Access List), 'Owner/Group/Other' y ACL (Access Control List),

en el que se identifica a los usuarios individuales o grupos de usuarios que pueden acceder a un archivo.

Particularmente, RBAC es un modelo que tiene diversas ventajas, ya que brinda a los equipos de TI la capacidad de agregar, modificar o quitar permisos a todos los usuarios de un grupo de una sola vez y de manera sencilla; o bien, de cambiar rápidamente el acceso de un solo usuario al asignarle o quitarle una función. Principalmente, RBAC disminuye los costos de gestión de usuarios en un sistema de información. Puede decirse que el valor de negocio de RBAC resulta de integrar seguridad y políticas de accesos con una vista organizacional de requerimientos de accesos. Esta integración es habilitada por tecnología, pero es además es una iniciativa de negocio que bien direccionada permite alinear políticas, procesos y responsabilidades organizacionales.

El potencial real de RBAC reside en la capacidad para gestionar permisos de acceso, o simplemente permisos, a través de muchas aplicaciones, redes y plataformas. Es decir, es posible con un mismo proceso RBAC controlar el acceso a una red corporativa, y además, por ejemplo, gestionar las tarjetas de acceso a edificios. De hecho, RBAC ofrece beneficios a toda la organización no tan solo a una aplicación específica.

Para que la implementación de RBAC en una organización sea exitosa, se necesita encontrar una manera de determinar el mínimo o más simple conjunto de roles que provea a los usuarios con todos los derechos de accesos y activos que ellos requieren, pero no más. Lo expresado está basado en el Principio de Mínimo Privilegio [4].

Actualmente, las organizaciones cuentan con más de un sistema de información implementados en tecnologías diversas y que se encuentran desplegados en diferentes servidores, sobre los cuales se necesita implementar políticas de control de acceso en diferentes niveles. Sin embargo, las soluciones de control de acceso única u "off-the-shelf" no siempre aplican en forma directa, por lo que se requiere de soluciones a la medida de las necesidades y características de las organizaciones.

En [5] se realizó un estudio sobre el modelo RBAC, y se realizó una adaptación y propuesta de diseño, así como una prueba de conceptos utilizando el paradigma de orientación a objetos. En este trabajo se retoma el modelo RBAC propuesto y se define un esquema de implementación de permisos inicial para el manejo de usuarios y roles en un sistema de ejemplo. Para mostrar la facilidad de su aplicación, se hará foco en la implementación del modelo propuesto en un entorno productivo.

A continuación se describe la estructura del trabajo. En la sección 2 se presenta el modelo RBAC propuesto empleando UML. En la sección 3 se presenta el caso de estudio en el que estará basado el esquema de usuarios, roles y permisos a implementar. En la sección 4 se presenta la implementación llevada a cabo empleando tecnologías modernas para desarrollo web. Finalmente en la sección 5 se presentan las conclusiones.

2 Modelo Role Based Access Control propuesto

El modelo RBAC incorpora el concepto de "rol" entre un sujeto (usuario/sistema) y el objeto protegido (dato o función), desacoplando los permisos entre el sujeto y el objeto correspondiente. El rol es el que tiene los permisos requeridos.

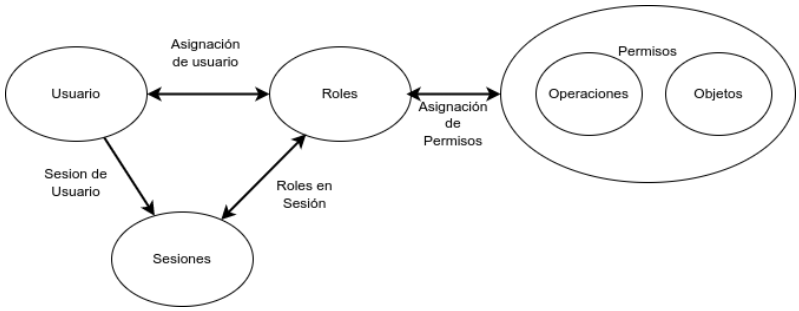

Fig. 1. Principales conceptos que intervienen en el modelo RBAC [2].

Un rol representa un conjunto de relaciones muchos a muchos entre usuarios individuales y permisos. Cada sesión (*Session*) es un mapeo entre un usuario y un subconjunto de roles activos que son asignados al usuario en un momento determinado. Un usuario puede tener muchas sesiones, y cada sesión tiene un subconjunto de roles asignados.

La idea básica de este modelo es asignar: (1) permisos a roles, y luego, (2) los roles son asignados a usuarios.

De esta manera, en una sesión, los usuarios pueden obtener los permisos de acceso a través de sus roles. La relación entre los elementos es: un usuario puede tener múltiples roles, un rol puede ser asignado a múltiples usuarios; un rol puede tener múltiples permisos, un permiso puede ser asignado a múltiples roles. Un "rol" es una función laboral dentro del contexto de una organización con cierta semántica asociada en relación con la autoridad y responsabilidad conferida al usuario asignado a tal rol. Un permiso es una aprobación para realizar una operación sobre uno o más objetos protegidos.

La Fig. 1 ilustra las relaciones de asignación de usuario y asignación de permiso. Las flechas indican una relación de muchos a muchos (por ejemplo, se puede asignar un usuario a uno o más roles y se puede asignar un rol a uno o más usuarios). Esta configuración proporciona una gran flexibilidad y granularidad en la asignación de permisos a roles y usuarios a roles. Esto evita que a un usuario se le pueda otorgar más acceso del necesario a los recursos.

Cada sesión es una asignación de un usuario a posiblemente muchos roles, es decir, un usuario establece una sesión durante la cual activa algún subconjunto de roles que se le asignan. Cada sesión está asociada a un único usuario y cada usuario está asociado a una o más sesiones. Los permisos disponibles para el usuario son los permisos asignados a los roles que están actualmente activos en todas las sesiones del usuario.

2.1 Representación del Modelo RBAC utilizando Orientación a Objetos

El modelo RBAC va a ser implementado utilizando el paradigma de Orientación a Objetos, y para esto se representa el modelo utilizando UML y Diagramas de Clase. En primer lugar, se propone un modelo de clases UML que sirve de base para la implementación del modelo RBAC en un sistema real (Fig. 2).

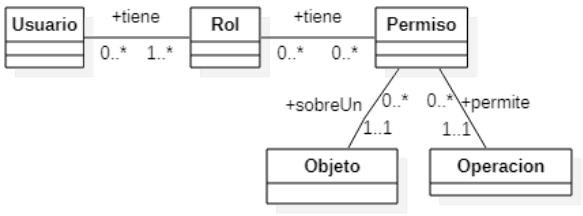

Fig. 2. Modelo de clases RBAC Base.

Agregado de restricciones de mejoras del modelo RBAC. El modelo RBAC considera la inclusión de restricciones para "separación de funciones" (segregation-of-duties, SoD). Las reglas estáticas de SoD (SSD) especifican que dos roles conflictivos específicos no pueden ser asignados al mismo usuario. Por ejemplo, un empleado de 'Cuentas a Pagar' no puede además procesar 'Cuentas a Cobrar', y el mismo individuo no puede ingresar y aprobar cheques recibidos. SSD define y ubica restricciones sobre el espacio total de permisos del usuario. La política de SSD es complementaria y debe ser utilizada al momento de asignar roles. SSD reduce la cantidad de permisos potenciales que pueden ser disponibles a un usuario mediante la definición de restricciones a ser aplicadas al momento de asignar roles.

La Separación de Funciones Dinámicas (Dynamic Separation of duty (DSD)) son incorporadas para limitar los permisos que están disponibles para un usuario. Pero lo hacen un contexto diferente a SSD: establece restricciones sobre los roles que pueden ser activados con o entre sesiones de usuario. Esto permite definir que un usuario tiene diferentes niveles de permisos en diferentes tiempos. Estas propiedades aseguran que los permisos no persisten más allá del tiempo necesario para realizar la función requerida. Los usuarios son autorizados a uno más role (SSD) pero están restringidos a ser activados simultáneamente. Por ejemplo: un usuario que está autorizado a tener roles 'Cajero' y 'Supervisor de Cajeros', si el usuario está en rol 'Cajero' no puede cambiar al rol 'Supervisor de Cajeros' hasta tanto no salga de su rol 'Cajero'.

Un elemento adicional al modelo base RBAC es el que denominamos 'Restricciones sobre Permisos'. Una restricción sobre un permiso permite definir un caso de granularidad más fina sobre un permiso asignado sobre un objeto protegido.

Table 1. Tipos de restricciones.

Restricción	Descripción
SOLO_LAS_PROPIAS	Puede aplicar la operación, pero SOLO para aquellas instancias en las que está relacionado directamente (sea el dueño de la instancia).
INSTANCIAS_HABILITADAS	De las instancias existentes, sólo puede aplicar el permiso sobre las instancias indicadas.
SOLO_LAS_DEL_AREA	Sólo puede acceder a las instancias del sujeto que pertenecen al área a la pertenece el usuario con este rol.

Un ejemplo típico de este tipo de restricciones es el de un usuario que tiene permiso de 'modificar' un objeto 'O', pero sólo aquellas instancias que él haya creado (SOLO_LAS_PROPIAS). En la implementación propuesta se incorpora como una

'Restricción' sobre un 'Permiso'. En la Tabla 1 se listan casos típicos de restricciones y la descripción de cada una, tales como las restricciones sobre instancias habilitadas (INSTANCIAS_HABILITADAS) y las restricciones sobre instancias del área al que pertenece el usuario o sujeto (SOLO_LAS_DEL_AREA). Este listado no es exhaustivo, siendo posible definir nuevos tipos de restricciones.

Por otra parte, el modelo RBAC considera el caso de que pueda darse 'herencia de roles', donde los roles con mayores permisos van heredando de roles con menores permisos. En la Fig. 3 se observa un ejemplo en instancias de roles aplicando herencia.

Aunque RBAC es altamente efectivo, implementarlo puede presentar desafíos, como la configuración y gestión inicial, la gestión continua de roles y asegurarse de que los derechos de acceso sigan alineados con los cambios en los roles y responsabilidades dentro de una organización.

Fig. 3. Ejemplo de instancias de roles.

Incorporando todas las definiciones anteriores al modelo base de RBAC, el modelo de diseño inicial a implementar queda de la forma en que se presenta en la Fig. 4.

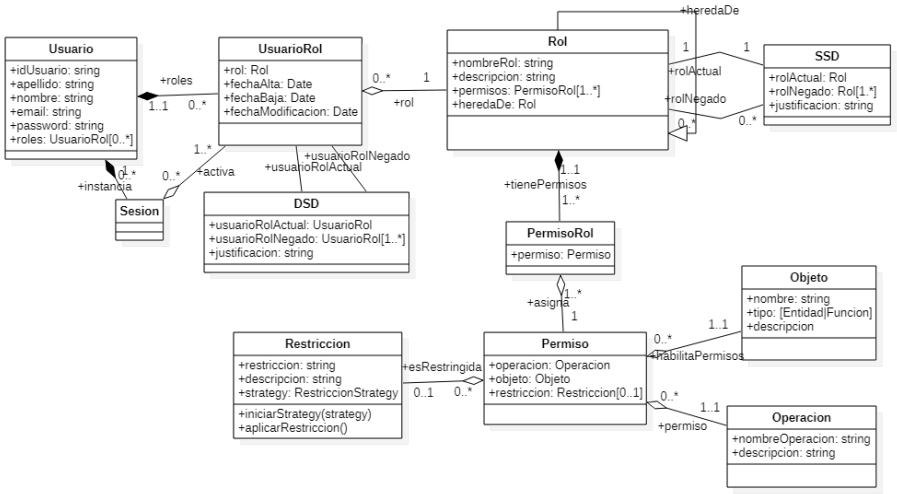

Fig. 4. Modelo de clases RBAC extendido.

3 Caso de estudio

Para probar la aplicación de RBAC a un caso de estudio, se consideró un sistema de compras, en el cual se tienen que gestionar las definiciones de 'Artículos', los que son clasificados en 'Rubros', siendo estos rubros son provistos por determinados 'Proveedores'. El modelo de clases en UML se vería como el mostrado en la Fig. 5.

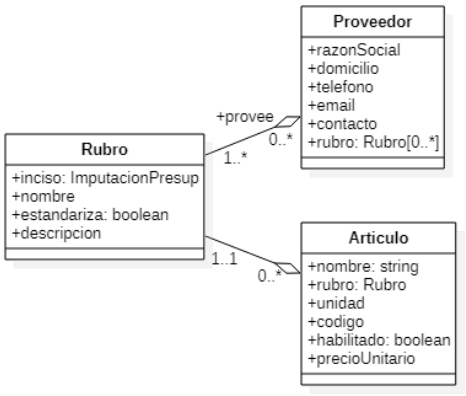

Fig. 5. Modelo de clases para la prueba de concepto.

Para poder aplicar RBAC es necesario definir las operaciones que se pueden aplicar sobre un objeto protegido. Para este ejemplo, las principales operaciones se presentan en la Tabla 2, las cuales se corresponden con las necesarias para realizar un CRUD (Create, Read, Update, Delete) sobre una entidad.

Table 2. Tipos de restricciones.

Operación	Descripción
Agregar	Agregar una instancia de una entidad.
Modificar	Modificar una instancia de una entidad.
Borrar	Borrar una instancia de una entidad.
Consultar	Consultar una/s instancia/s de una entidad.

Luego, los 'Permisos' sobre un 'Objeto' se dan estableciendo la relación 'Objeto / Operación' permitidos. A modo de ejemplo se indican en la Tabla 3 un subconjunto de permisos para el sistema.

Table 3. Tipos de operaciones.

Objeto	Operación	Permiso
Articulo	Agregar	Agregar_Articulo
	Modificar	Modificar_Articulo
	Borrar	Borrar_Articulo
	Consultar	Consultar_Articulo
Rubro	Agregar	Agregar_Rubro
	Modificar	Modificar_Rubro

	Borrar	Borrar_Rubro
	Consultar	Consultar_Rubro
Proveedor	Agregar	Agregar_Proveedor
	Modificar	Modificar_Proveedor
	Consultar	Consultar_Proveedor
	Borrar	Borrar_Proveedor

3.1 Definición de Roles y Permisos

Para el caso de estudio se definen los roles y permisos correspondientes, los que se muestran en la Tabla 4.

Table 4. Roles y permisos por Objeto.

Rol	Descripción	Permiso	Objeto
Administrador	Administra el sistema. Puede realizar ABMC de todas las entidades, y funciones de administración.	Tiene TODOS los PERMISOS sobre TODAS las entidades propias del sistema.	Proveedor Rubro Artículo
Vendedor	Es la persona/usuario que puede realizar 'Ventas de Artículos'.	Consultar_Proveedor	Proveedor
		Consultar_Rubro	Rubro
		Consultar_Artículo	Artículo
		Modificar_Artículo	Artículo
Evaluador Técnico	Es la persona/usuario que puede evaluar las definiciones de artículos que pertenecen a los rubros que él evalúa.	Consultar_Artículo	Articulo
		Modificar_Artículo	Articulo
		Agregar_Artículo	Artículo
		Eliminar_Artículo	Artículo

4 Implementación en un entorno operativo

Para realizar la prueba de conceptos en un entorno de ejecución real, se utilizaron los siguientes paquetes de software: el lenguaje de programación web PHP versión >= 7.2.5, el framework para desarrollo de aplicaciones web Symfony 5.4.*, Doctrine 2.11 como mapeador objeto-relacional (ORM) que proporciona una capa de persistencia para objetos Symfony/PHP, y una base de datos de Mysql 8.0.

Symfony es un framework de desarrollo de aplicaciones Web basadas en PHP 7 o superior, que provee un amplio abanico de herramientas que facilitan el desarrollo de aplicaciones monolíticas o basadas en servicios. PHP es un lenguaje de código abierto especialmente adecuado para el desarrollo web, y que puede ser incrustado en HTML, además implementa el paradigma de Orientación a Objetos con todos sus elementos. Mysql es un gestor de bases de datos relacionales ampliamente conocido, y el ORM Doctrine hace de nexo entre Symfony/PHP y el gestor de Bases de Datos, por lo que este podría ser remplazado por otro gestor de bases de datos relacional con mínimo esfuerzo.

Todas las herramientas seleccionadas son de código abierto y multiplataforma. Además, para las pruebas se selecciona un entorno basado en Linux, pero puede fácilmente ser replicado en un entorno basado en Windows.

Implementación en Symfony. Una vez instalado todo el entorno de desarrollo, se procede a crear las entidades y elementos de software necesarios para la prueba de conceptos. En la Fig. 6 se pueden ver las entidades creadas para el modelo RBAC (prefijo 'Rbac') y las entidades del problema (*Articulo*, *Proveedor*, *Rubro*). El archivo '*User.php'* representa al usuario tal como se define en el framework utilizado, del cual se comentará posteriormente su implementación.

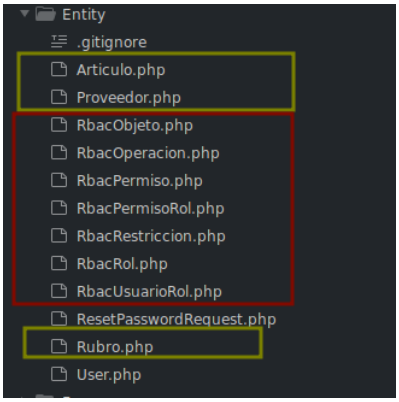

Fig. 6. Modelo de Clases para la prueba de concepto.

En la Fig. 7 se puede observar una vista parcial de cómo se implementa la clase 'Permisos' (*RbacPermiso.php*).

```
class RbacPermiso
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\ManyToOne(targetEntity=RbacOperacion::class,
     inversedBy="rbacPermisos")
     * @ORM\JoinColumn(nullable=false)
     */
    private $rbacOperacion;

    /**
     * @ORM\ManyToOne(targetEntity=RbacObjeto::class,
     inversedBy="rbacPermisos")
     * @ORM\JoinColumn(nullable=false)
     */
    private $rbacObjeto;

    /**
     * @ORM\ManyToOne(targetEntity=RbacRestriccion::class,
     inversedBy="rbacPermisos")
     */
    private $rbacRestriccion;
```

Fig. 7. Vista parcial de la clase *RbacPermiso*.

Y en la Fig. 8, una vista parcial de clase *RbacUsuarioRol.php* que vincula un usuario con sus diferentes roles.

```php
/**
 * @ORM\Entity(repositoryClass=RbacUsuarioRolRepository::class)
 */
class RbacUsuarioRol
{
    *
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")

    private $id;

    /**
     * @ORM\ManyToOne(targetEntity=RbacRol::class,
     inversedBy="rbacUsuarioRols")
     * @ORM\JoinColumn(nullable=false)
     */
    private $rbacRol;

    /**
     * @ORM\ManyToOne(targetEntity=RbacUsuario::class, inversedBy="rbacRoles")
     */
    private $rbacUsuario;
```

Fig. 8. – Vista parcial de la clase que relaciona Usuarios y Roles (*RbacUsuarioRol*).

Implementación de lógica de RBAC. Para facilitar la utilización del control de acceso por los objetos que deben ser protegidos, se implementa la lógica de control de autorización como un 'servicio': RbacService.php. En la Fig. 9 se observa parte de este servicio.

```php
public function denyAccess(User $user, string $operacion,$object): void
{
    //Del parámetro $object se extrae la clase/objeto concreto que está siendo evaluado
    list($a,$b,$clase)= explode("\\",get_class($object));
    $autorizado=false;

    $objeto=$this->em->getRepository(RbacObjeto::class)->findOneBy(["nombre"=>$clase]);
    $objOperacion=$this->em->getRepository(RbacOperacion::class)->findOneBy(["nombreOperacion"=>$operacion]);

    if($objeto==null || $objOperacion==null){
        throw new AccessDeniedException("Acceso denegado.");
    }
    //Buscar el rol activo del usuario de la Sessión de Usuario
    $rolSelected = $_SESSION['rolSelected']; //El indice del rol seleccionado por el usuario
    $rolActivo = $user->getRbacRols()[$rolSelected]; //Objeto RbacRol activo

    $rol = $user->getRbacRols()[$_SESSION['rolSelected']];

    $permisos=$rol->getRbacPermisos(); //Recuperar los permisos del rol
    //Buscar en los permisos del rol si está el permiso requerido
    foreach($permisos as $permiso){
        if($permiso->getRbacPermiso()->getRbacObjeto()->getId()==$objeto->getId() &&
           $permiso->getRbacPermiso()->getRbacOperacion()->getId()==$objOperacion->getId()){
               $autorizado=true;
           }
    }
    if($autorizado==false){
        throw new AccessDeniedException("Acceso denegado.");
    }
```

Fig. 9. Definición del control de autorización *denyAccess* que forma parte del servicio *RbacService.php*.

Este servicio se 'inyecta' en cada controlador de objetos que requiera control de autorización. Por ejemplo, el siguiente bloque de código muestra cómo se inyecta el 'RbacService' en el controlador del objeto Rubro, y luego se invoca el servicio

'denyAccess' que realiza el control de autorización. Puede observarse que éste devuelve una excepción si el permiso está denegado.

```
class RubroController extends AbstractController
{
    /**
     * @Route("/new", name="app_rubro_new", methods={"GET", "POST"})
     */
    // Se inserta el servicio 'RbacService' en la lamada del método new
    public function new(Request $request, RubroRepository $rubroRepository,
RbacService $rbacService): Response     {
        $rubro = new Rubro();
        try{
            // Llamada al método 'denyAccess()' del servicio $rbacService, que
recibe como parámetros:
            // el usuario, la operación, y el objeto protegido
            $rbacService->denyAccess($this->getUser(),'AGREGAR',$rubro);
        }
        catch (\Exception $e){
            return $this->renderForm('shared/acceso_denegado.html.twig', []);
        }
        $form = $this->createForm(RubroType::class, $rubro);
...
}
```

Implementación de Usuarios. El concepto *Usuario* (clase 'Usuario' de la Fig. 4) es instanciado con los usuarios propios de la aplicación y en la implementación propuesta están representados en la entidad *'User.php'*. Como se observa en el modelo de la Fig. 4, un *Usuario* puede tener uno o más roles (concepto *Rol*), lo cual se expresa de manera explícita mediante la clase *UsuarioRol*. En Symfony utilizando el ORM Doctrine, esta relación se define de la siguiente manera, sindo la clase 'RbacUsuarioRol' la que implementa el concepto UsuarioRol:

```
        /**
         * @ORM\ManyToMany(targetEntity=RbacRol::class, inversedBy="users")
         * @JoinTable("RbacUsuarioRol")
         */
        private $rbacRols;
```

La clase 'User.php', es una clase que debe ser extendida incorporando los métodos necesarios para la gestión propia de los elementos de un '*RbacUser'*, como por ejemplo '*addRbaRol'* (agregar un Rol al usuario), '*getRbacRols'* (recuperar los Roles de un Usuario), y '*removeRbacRol'* (eliminar un Rol asignado a un Usuario). Los fragmentos de código relacionado son:

```
        public function addRbacRol(RbacRol $rbacRol): self    {
                if (!$this->rbacRols->contains($rbacRol)) {
                    $this->rbacRols[] = $rbacRol;
                }
                return $this;
            }

        public function getRbacRols(): Collection  {
                return $this->rbacRols; }
```

```
public function removeRbacRol(RbacRol $rbacRol): self     {
       $this->rbacRols->removeElement($rbacRol);
       return $this;}
```

Los aspectos de gestión de la seguridad relativos a autenticación y autorización inicial de usuarios se delegan a la implementación provista por el framework, por lo que se implementa el bundle '*SecurityBundle'* de Symfony.

Implementación de funciones básicas de RBAC. Las funciones básicas requeridas para RBAC se pueden reducir a dos casos de uso (Fig. 10). El caso de uso *LoginEnSistema* se modela en el diagrama de secuencia de la Fig. 11.

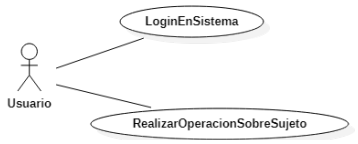

Fig. 10. Casos de uso que incluyen las funciones básicas requeridas por RBAC.

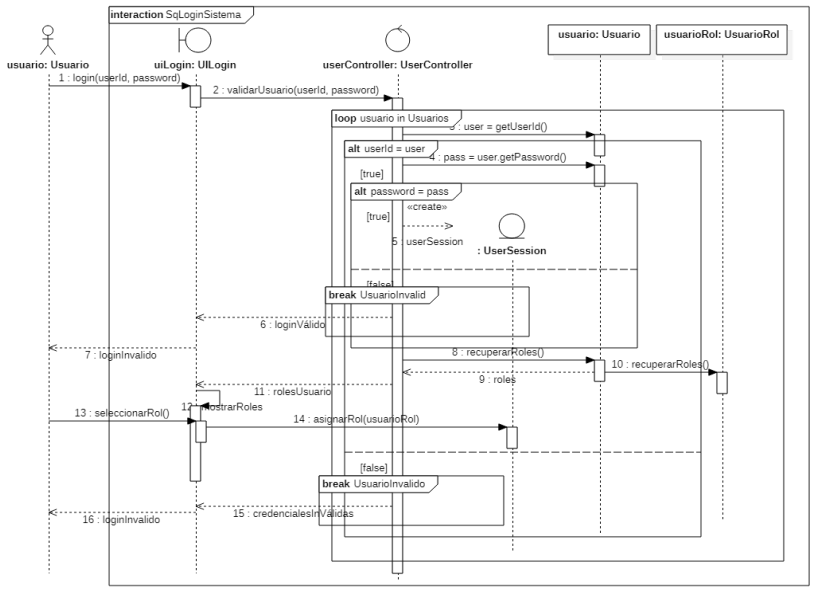

Fig. 11. Diagrama de secuencia inicial para la especificación del caso de uso *LoginEnSistema*.

En la Fig. 11 se describe que el '*Usuario'* ingresa sus credenciales y son validadas (pasos 1 a 4). Si las credenciales son válidas (userId y password válidos) se crea un '*Session'* de usuario (paso 5: Crear '*UserSession'*), y a continuación, se recuperan los roles asociados al usuario, enviando un mensaje al objeto *UsuarioRol* (pasos 8 a 10). Luego, el usuario debe seleccionar el rol con el cual va a trabajar (pasos 11 a 14). Finalmente, el rol seleccionado se asigna a la 'Session' del usuario, objeto *UserSession*, (paso 14, *asignarRol()*). En el caso de que las credenciales no son válidas, se informa al usuario y el proceso se cancela.

El comportamiento asociado al caso de uso *RealizarOperacionSobreSujeto*, el cual corresponde al control de autorización, se especifica en el diagrama de secuencias de la Fig. 12.

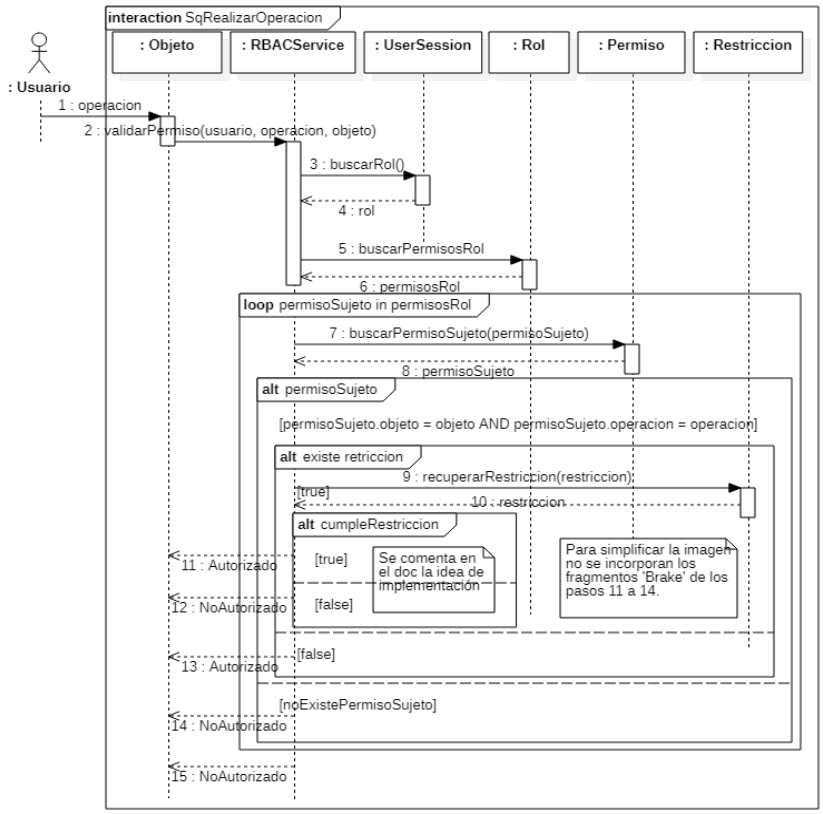

Fig. 12. Diagrama de Secuencia inicial para la especificación del caso de uso '*RealizarOperacionSobreSujeto'*.

Se observa que el 'Usuario' ya se encuentra logueado y requiere hacer una 'operación' sobre un 'Objeto' que representa un objeto protegido en el sistema (paso 1). El objeto invoca el método '*validarPermiso()*' del servicio '*RBACServic*e' (paso2). El servicio recupera el 'rol' activo del usuario que se mantiene en '*UserSession*' (paso 3), y busca los permisos asociados a ese rol (paso 5). Para cada uno de los permisos recuperados (loop *permisoSujeto in permisosRol*) busca el '*Permiso'* concreto referenciado (paso 7); con el permiso recuperado (*permisoSujeto*) verifica que este permiso sea para el objeto y operación requerido (*permisoSujeto.objeto = objeto AND permisoSujeto.operacion = operación*). Si ello se cumple, el usuario está autorizado para realizar la operación sobre el objeto, sino, es denegado.

Propuesta para implementar una restricción sobre un permiso. Como se comentó anteriormente, el concepto de *Restricción* no se encuentra definido explícitamente en el modelo RBAC estándar, pero fue incluido en el modelo extendido propuesto en este

trabajo, ya que se trata de una situación que se presenta con frecuencia en la práctica. Sin embargo, al momento de la implementación del modelo, se presenta el problema de que tiene la aplicación de una restricción sobre un permiso depende del contexto en donde se aplica, y no puede definirse en forma genérica para todos los dominios.

La solución propuesta consiste en aplicar el patrón de diseño de comportamiento '*Strategy'* [7] , como se observa en la Fig. 13.

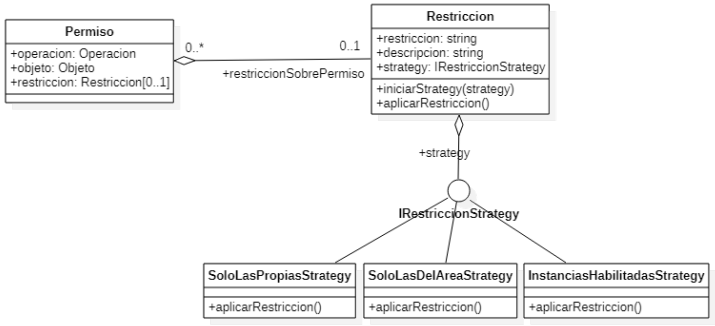

Fig. 13. Aplicar un patrón *Strategy* para resolver las definiciones de Restricción

Dicho patrón lo que propone es definir una interfaz '*IRestriccionStrategy'* (la estrategia) que la clase '*Restriccion'* referencia como interfaz genérica de su operación '*aplicarRestriccion*()'. Luego, se proponen tres clases (que pueden ser extendidas a todas las necesarias) que deben implementar la operación concreta '*aplicarRestriccion()*' de acuerdo a lo que esta signifique en el contexto particular. Las clases '*SoloLasPropiasStrategy*', '*SoloLasDelAreaStrategy*', e '*InstanciasHabilitadasStrategy*' deben ser implementadas apropiadamente en cada instalación para una organización particular.

Auditoría y loggin de actividad. La auditoría y registro de actividad (loogin) se propone en dos niveles:

a) Interno de RBAC. Incluyendo registro de auditoría sobre las operaciones realizadas sobre las entidades propias de RBAC, por ejemplo, agregar, modificar o eliminar un Permiso; y registro de actividades en el uso de RBAC. Por ejemplo, se invoca el servicio para RealizarOperacionSobreSujeto y en ese caso se deja un log de auditoría con la información sobre quién y qué operación realizó. b) A nivel aplicación. Donde la aplicación propiamente debe implementar los registros de Auditoría sobre sus entidades. Por ejemplo, agregar, modificar o eliminar un Artículo. Y los logs de actividad que considere necesario.

En Symfony eso se puede lograr utilizando auditor-bundle [11] y monolog-bundle [12].

5 Conclusiones

Se estudió el modelo RBAC y trabajó en el diseño de un modelo específico para un sistema de compras genérico. Esto significó llevar adelante las actividades de análisis necesarias para la identificación de usuarios, entidades, permisos, y roles requeridos.

Además, se definieron algunas restricciones para hacer más específicos los permisos a activar en determinadas sesiones, dependiendo de las relaciones del usuario con respecto a la propiedad o no sobre entidades del sistema.

La implementación como prueba de concepto permitió comprobar que el modelo RBAC adaptado en esta propuesta, puede ser implementado trabajando con un framework una configuración básica de tecnologías para desarrollo web.

Como trabajos a futuro se extenderá el modelo de diseño incluyendo la posibilidad de 'herencia de roles'', 'restricciones', 'SSD', 'DSD, y también la correspondiente implementación de un proceso de asignación automática de roles en un caso de uso extendido. Además, se prevé evaluar las ventajas del enfoque y comparar con otros en cuanto a la facilidad de su implementación, la agilidad en gestión de usuarios, y la facilidad de incorporación de cambios, entre otros. Por otra, debe también contemplarse los procesos de Gestión de Usuarios en cuanto a la asignación y modificación de permisos a los usuarios de acuerdo a las políticas definidas al respecto por la organización. Finalmente, debe analizarse cómo extender al servicio para que pueda ser utilizado (como única instancia) por múltiples sistemas en una organización.

Referencias

1. International Committee for Information Technology Standards, ANSI INCITS 359-2012 Information Technology - Role Based Access Control (2012).
2. Mohamed, A.K.Y.S., Auer, D., Hofer, D. and Küng, J. (2022), "A systematic literature review for authorization and access control: definitions, strategies and models", *International Journal of Web Information Systems*, Vol. 18 No. 2/3, pp. 156-180. https://doi.org/10.1108/IJWIS-04-2022-0077
3. Gasser, M.: Building a secure computer system. Van Nostrand Reinhold Co., USA (1988)
4. Bishop, M.: Computer Security: Art and Science. Addison-Wesley Professional, Boston, MA (2003).
5. Ramos, J. C., Roldan, M. L. Definición de esquema de asignación de permisos. Actas de la I Jornada de Ciberseguridad y Sociedad. *AJEA (Actas De Jornadas Y Eventos Académicos De UTN)*, AJEA 27, pp. 17-24 (2023).
6. Qubera Solutions, Inc. *Implementing Role-Based Access Controls in the Enterprise* (2011)
7. Gamma, E., Helm, R., Johnson, R.,, Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. ISBN: 0201633612
8. Symfony (2024), Installing & Setting up the Symfony Framework, Symfony, https://symfony.com/doc/5.x/setup.html
9. Doctrine (2024), Databases and the Doctrine ORM, Symfony, https://symfony.com/doc/5.x/doctrine.html
10. PHP (2024), Documentation, PHP, https://www.php.net/docs.php
11. Auditor Bundle (2024), https://damienharper.github.io/auditor-docs/docs/auditor-bundle/index.html
12. Monolog Bundle Component (2024), https://symfony.com/components/Monolog%20Bundle