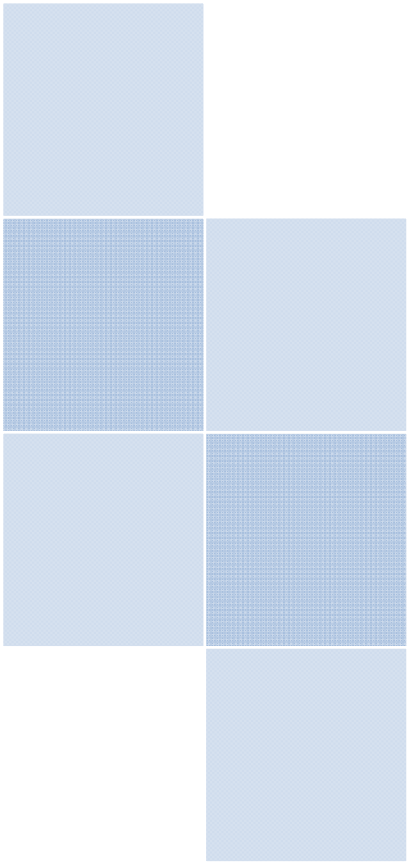
Role-Based Security in a Hierarchical Environment

A conceptual and practical exploration

Bas Geertsema
b.geertsema@xihsolutions.net



Abstract

We explore the approaches that current security frameworks take in software development. Some approaches are too basic to be useful in more complex applications. After identification of the requirements of a more advanced framework, a conceptual framework is developed with the distinct feature that a role is bound to a *context* within a *hierarchy*. It resembles the security frameworks found in traditional file systems. This design provides a fine-grained and flexible control of permissions. An implementation is developed in Microsoft SQL Server 2005 based on this conceptual framework. The implementation is extended with extra features for fast performance. A full specification of the implementation is available.

Keywords

software development, design, security, framework, sql, role-based, tree, hierarchy, sql server 2005

Version 1.0 / first public release, October 2007

Contents

Designing a security framework

Introduction

Every application that grows beyond your own personal domain will at some point face the need to deal with security issues. With the arrival of the internet and the movement of moving applications online this has only become more important. The security framework is one of the foundations of an application and must be considered in all stages of the software development process. An implementation should be embedded in any software project as early as possible. In this paper we will explore security frameworks from a conceptual point of view first. This allows us to look at different approaches and the differences between them. In the second part of this paper we will develop a powerful and generic security framework based on these insights and provide an implementation developed in MS SQL Server 2005.

Approaches to a security framework

In the following section we will take a high level view on the concept of security frameworks, how we defined it and the different approaches that can be taken in the design of a framework.

Definitions

Throughout this paper, we often refer to security related terms. We start off by defining these terms so it is clear what we understand by these terms:

1. **System** – A system is the application, or collection of applications, that offers the functions to be performed. For example a web-based application such as a forum, that offers the function to post a reply.
2. **Function** – An action that can be performed by the system and that adds value to the system. For example the function to post a reply on an online forum**.**
3. **Principal** - The identity that is authorized to perform functions. Note that a principal can be a regular user, but might as well be another system.
4. **Role** – The representation of the possibility to perform a set of functions. A principal can be a member of multiple roles.

These four elements constitute the foundation of most security paradigms and are the building blocks for the security framework we will develop. A security framework defines how these elements are designed and how they work together to achieve the security harness for a system.

Workings of a framework

How can we use the concepts of systems, functions, principals and roles to create a security framework? Most of the frameworks boil down to the following approach:

1. The user or application is represented by a principal, or token.
2. An authentication process identifies the principal.
3. Before a function is performed, the security framework authorizes the principal to perform this function. Depending on the outcome of the authorization the function is executed or not.

A traditional and intuitive setup of a security framework that follows from this is to:

1. Define the distinct functions in the system
2. Define the roles
3. Define the set of functions that each role is allowed to perform
4. Assign roles to principals

In many cases, this is the easiest and best approach to take. However, for some systems this is just not flexible enough and does not cover the requirements. One disadvantage of this traditional framework is that the roles are *context unaware*. This means that a principal is either in a role, or it is out. And when it is in, it has access to *all* functions and resourced allowed by that role. It does not take in account upon which part of the system a function performs. In short, it is *not* possible to partition your security system beyond the initial partitioning based on roles. In some systems there *is* a need for extra partitioning. For example, most ERP (Enterprise Resource Planning) systems will provide partitioning based on customer. Or within a big organization, a system can be partitioned in several organization units. In these cases, an account manager will only be able to see the customer he or she is assigned to. And the HRM manager at location A will not be able to perform functions on employees at location B.

In these setups the security frameworks faces a new challenge: how to incorporate both the role-based partitioning *and* the system partitioning. A solution is to make the roles what we call *context-aware*. This means that a role is applicable, based on role *and* the *context* the function is performing on. For example, imagine you are developing a forum web application. A forum can consist of multiple sub forums. You want to embed the possibility to assign different moderators for different sub forums. Quickly, you figure out that using pure role-based security will not work in this case. You setup a scheme where you assign a pair of both role *and* sub forum to your moderators. In this case, the sub forum is the *context*. Context-aware role based security frameworks are very common. Just think of file systems where roles are assigned to files. In this paper, we want to take this framework one step further, and introduce the concept of a *hierarchy*.

Put on your architectural glasses, and look to the world outside. Suddenly you will start to see hierarchies everywhere. On your work, you see regular employees, managers, bosses and bosses-bosses. Public governance is partitioned in central government, state government, city government, neighborhood council all the way down to the governance of your own household. Maybe you own a house, in which you have different rooms, in which you have furniture. When you want to browse to a picture on your computer you will start at *My Documents*, browse down to *My Pictures*, and see a list of the pictures you own. A variety of hierarchies, but what relates them even more is the fact that

when you have power up in the hierarchy, you will have power all the way down the hierarchy. Your state government controls the budget that will go to the city government, going down to your neighborhood council. If you own a house, you are most probably also owner the interior. If you own your *My documents* folder, you will own your *My Pictures* folder.

As it turns out, a lot of security frameworks do not only work on role-context basis, but on a *hierarchical, contextual and role-based* basis. This means the roles are assigned to a context object, and this role is automatically applied to all descendants of this context object. A good example of this kind of security framework is a traditional file system, where roles assigned to a directory are often automatically applied to all underlying files and directories. In the next section we will develop a framework that features these properties: *hierarchical, context-aware* and *role-based*.

Developing the conceptual framework

In this paper we will outline the conceptual features of such a framework. We will outline the use case for the security framework we will be developing in the course of this paper. This is based on a real-world application that we have developed recently. A requirement of the application was that it could serve multiple clients in an isolated fashion. The application controls human resources and it closely mirrors the actual organizational structure of the client. For example, a client can define its organization as one headquarter that contains four physical locations. In turn, each location can contain one or more functional organization units (sales, logistics, etc). The organization units contain the employees.

For each client we want to define an administrator that is able to perform a variety of functions that will configure the application *within* the boundaries of the client. We also want a system administrator, who is able to execute modification functions on the whole system, and for all clients. Because some data is commonly shared between clients, we want all this data in a single database and provide partitioning on the application level.

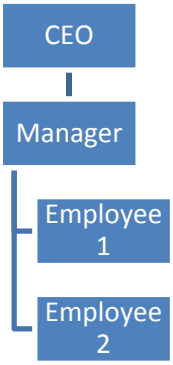
Data model

We start off by defining our hierarchy. For the experienced object oriented developer, the composite pattern will come to mind. In short, this means that we compose a tree by linking uniform objects that can be either leafs (containing no children) or containers (containing children). Even though you may have a mix of types in your tree, they all must implement the composite interface. This way you can treat every object in the tree in a uniform way, which will allow for very abstract and general code. In our case the common denominator is that every object can be considered as an *organization unit*. An organization is an organization unit, a user is an organization unit and a physical location is an organization unit. So we can neglect the differences between these kinds of organization units, and from now on we will only refer to organization unit, even though in reality they might be other types.

Note that we could have abstracted away the organization units and implement the framework more generic as working on a tree structure of nodes. And thereby not defining what those nodes are. However, we feel that translating generic and abstract *nodes* to the actual case of *organization units* makes it easier to grasp the workings of the framework.

The most straight-forward for representing a tree in a relational data model is by setting up a 1:n, or a one-to-many relation between child and parent. Consider the following hierarchy:

ID	Name	ParentID
1	CEO	*Null*
2	Manager	1
3	Employee 1	2
4	Employee 2	2



The table on the left represents the hierarchy on the right. In this case, the manager reports to the CEO and both employees report to the manager.

Because of the mismatch between the paradigm of a relational structure and that of a hierarchical structure, there are more ways of representing a hierarchy in a relational structure. We will come to that later on, when we will start optimizing the structure. However, as we will see, this setup is the most intuitive and robust way and will form the foundation upon which you will build extra functionality to overcome performance problems. Storing a tree in this fashion is known as *adjacency lists*.

Authentication and authorization

We defined the data model to represent the hierarchy. Now we will make this model useful by providing authentication. First the difference between authentication and authorization is explained. There is not a fixed definition on authentication, but the following definition[1] covers it quite well:

Authentication is the process of determining whether someone or something is, in fact, who or what it is declared to be.

In many applications, this is being done through a username/password combination. But it might as well be biometric identification or voice recognition. I will not go further into authentication since it is out of the scope of this paper and it differs between different applications. However, I will assume that after a successful authentication the result is a *principal*. A principal is the identity upon which authorization is performed. A principal can be a user, but it might be an application as well. Because of this abstraction it is preferred not to talk about *users*, but about *principals*.

Here follows a definition of authorization[2]:

Authorization is the process of giving someone permission to do or have something.

Authorization is the part we are the most interested in and that we will explore more in this section. The definition above is very high-level. There are many ways in achieving the authorization process, some of which we have explained earlier in our discussion of approaches to a security framework. A pure role-based approach turned out to be too simple to meet our requirements. The framework we develop performs authorization based on *roles* and a *context* in a *hierarchical* structure. In effect this means a principal is assigned a role with the following properties:

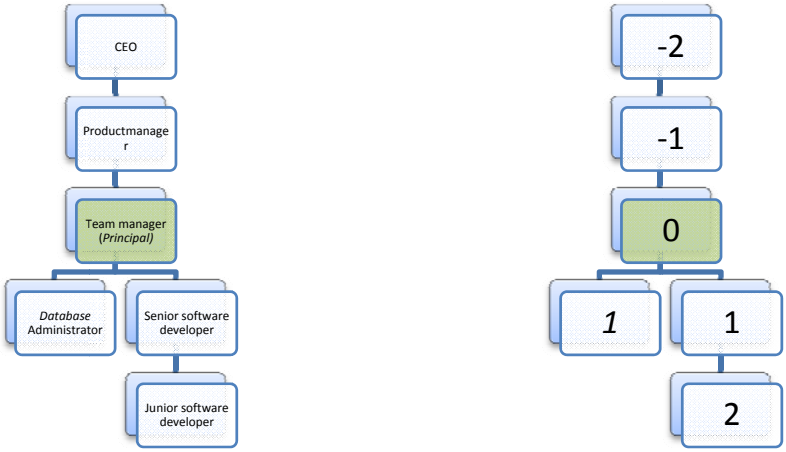
[1] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci211621,00.html
[2] http://www.bitpipe.com/tlist/Authorization.html

- Role code
- Context
- Minimum pathlevel
- Maximum pathlevel

Where Minimum pathlevel <= Maximum pathlevel. Pathlevel is relative, which means the following:

- All ancestors (parents, parents of parents, etc) of the context has relative pathlevel < 0
- The context itself has relative pathlevel 0
- All descendants (children, children of children, etc) of the context have pathlevel > 0

The properties or the role entity define the *function it can perform*, the *context* and the *coverage* of the *role*. The following diagram should make it clearer.



Figuur 1O Hierarchy structure (left) with relative pathlevel (right). The context is marked green.

For example, let's assume there is a role *SalaryRapport* which gives the principal insight in salary details of an employee. In most cases, a manager is allowed to see its subordinates' salaries. And in general the manager is not allowed to see the salaries of its bosses. So the role *SalaryRapport* would have the parameters:

```
RoleCode=SalaryRapport, Context=manager, MinPathLevel=0, MaxPathLevel=100
```

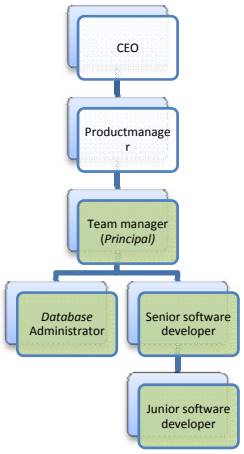
Where 100 is an arbitrary large number (In most cases you probably do not want a hierarchy nesting 100 levels, but feel free to put the number up). This means the role is applicable to the context itself (relative pathlevel = 0) and all descendants (relative pathlevel > 0). In the same way, roles that are only applicable to the context have MinPathLevel=0 and MaxPathLevel=0. The following table lists common cases with the applicability of the role:

MinPathLevel	MaxPathLevel	Applicable to
0	0	Context
0	Max	Context and all descendants
1	Max	All descendants of context
-Max	0	Context and all ancestors
-Max	-1	All ancestors of context
1	1	Only direct children of context

This approach of defining roles and role applicability gives you enormous flexibility and works in an intuitive way. We will show some examples of roles. Let's assume we have identified the following role

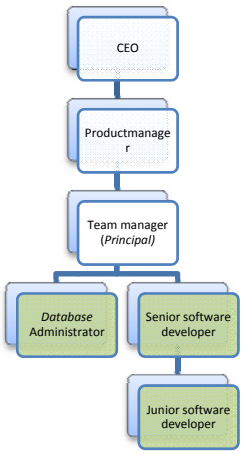
```
Role = {Context = Team Manager, RoleCode = SalaryRapport, MinPathLevel = 0, MaxPathLevel = 100}
```

The principal is allowed to execute the function specified by the role on the organization units marked green in the following diagram:



Let's resume with another example that is a bit more complicated. Assume there is a role *ShowEmployeeDetails* which shows the obvious person details like name, address, telephone number, etc. You want the team members to be able to see the user details for their other team members (peers) and their descendants, but not from the team manager itself (ancestors). You can define the following role:

```
Role = {Context = Senior software developer, Context = Team Manager, RoleCode = ShowEmployeeDetails, MinPathLevel = 1,MaxPathLevel = 100}
```



So the trick here it that you actually apply the role to the 'team manager', but since MinPathLevel is 1 the team manager itself is not included, but all its direct children (and thus the siblings of the senior software developer) and their descendants are. Any sibling that is now added as a child of Team Manager is automatically with the coverage of this role.

We conclude that a role is not only bound to a single organization unit, but conveys *multiple* organization units. And this set of organization units is adaptive, which means that new organization units inserted will automatically be under the coverage of the role, if it is applicable. You define a role just once on a single unit, and the tree structure will determine which units are in the coverage set.

We have finished the conceptual part of our security framework. Although these concepts are quite simple and straightforward, it offers a lot of flexibility and power in your authorization process. In the next section we will look into an implementation.

An implementation of the framework

Our implementation is based on Microsoft SQL Server 2005, but one should be able to transfer it to other DBMS suppliers. There is one feature however that might not be available in other database servers. These are Common Table Expressions (CTE), which can be used to execute recursive queries. An equally valid construct is available in Oracle databases. The lack of recursive queries might be a limitation, but in the next section we will optimize the framework by leaving out the CTEs in the core functionality and take a different and more vendor independent approach on traversing a tree structure.

The implementation that we will put forward in this paper is by no means the only right solution. It is an implementation that we find very practical to work with and performs well. It offers us all the flexibility and features that we need. Always consider whether you have the same case, and feel free to tweak parts of the framework to fit your needs.

Data model

We start off by defining our tree structure.

Table 1 OrganisationUnit

Name	Type	Allows NULL
Id	int (primary key, identity increment)	No
ParentId	int	Yes
Name	varchar (100)	No

Table 2 Role

Name	Type	Allows NULL
PrincipalId	int (primary key, identity increment)	No
Code	varchar (50)	No
MinPathLevel	int	No
MaxPathLevel	int	No
OrganisationUnitId	Int	No

Test data OrganisationUnit

Id	ParentId	Name
1	*NULL*	CEO
2	1	Product manager
3	2	Team manager
4	3	Database administrator
5	3	Senior software developer
6	5	Junior software developer

Test data Role

PrincipalId	Code	MinPathLevel	MaxPathLevel	OrganisationUnitId
1	ModifyUserDetails	0	100	1
2	ViewProjectStatus	0	0	2
3	AssignTaskToUser	0	100	3

4	AskUserForPayRaise	-1	-1	4
5	AssignTaskToUser	0	100	5

The meaning of the roles in the test data can be easier to understand in natural language:

- The CEO is allowed to modify user details for itself and all its descendants (that is the whole company)
- The project manager has a traditional role to view status of projects. This role is bound to the person itself, and is not working down- or upwards. It would make no sense to do so.
- The team manager is allowed to assign tasks to itself and its descendants (the whole team).
- The database administrator is allowed to ask for a pay raise to its direct parent (the team manager).
- The senior software developer is allowed to assign tasks to itself and its descendants (the junior software developer).

The following step is to specify how an application can deduct from this data whether a given action is permitted for a given principal and a given context organization unit. First we create a database view that will help us mapping the tree in such a way that we can use our regular relational SQL operations on it.

```
WITH OrganisationAncestor AS
  (SELECT Id, Id AS ContextId, ParentId, 0 AS PathLevel
         FROM dbo.OrganisationUnit
    UNION ALL
   SELECT a.Id, b.Id AS ContextId, b.ParentId, a.PathLevel - 1 AS PathLevel
         FROM dbo.OrganisationUnit AS b
         INNER JOIN OrganisationAncestor AS a ON a.ParentId = b.Id),

OrganisationDescendant AS
    (SELECT Id, Id AS ContextId, 0 AS PathLevel
         FROM dbo.OrganisationUnit AS c
     UNION ALL
     SELECT d.Id, e.Id AS ContextId, d.PathLevel + 1 AS PathLevel
     FROM dbo.OrganisationUnit AS e INNER JOIN
         OrganisationDescendant AS d ON d.ContextId = e.ParentId)

SELECT  Id, ContextId, PathLevel FROM OrganisationAncestor AS Ancestor
UNION
SELECT  Id, ContextId, PathLevel FROM OrganisationDescendant AS Descendant
```

This view uses two common table expressions. The first CTE is used to recursively traverse from a given organization unit to all its ancestors. The second CTE is used to recursively traverse from a given organization unit to all its descendants. The result is a list that can be interpreted as follows:

A combination of each organization unit, with all its ancestors, itself and its descendants. A relative pathlevel is calculated with the origin (0) being the organization unit itself. Positive being downwards in the tree and negative being upwards in the tree.

An example of this would be for the team manager:

PrincipalId	ContextId	PathLevel
3 (Team manager)	1 (CEO)	-2
3 (Team manager)	2 (Project manager)	-1
3 (Team manager)	3 (Team manager)	0
3 (Team manager)	4 (Database administrator)	1
3 (Team manager)	5 (Senior software developer)	1
3 (Team manager)	6 (Junior software developer)	2

The result of this view can become huge very fast in even modest tree sizes, but you will never need the whole view and filter out most of it, which makes the performance acceptable. This view will be a very utile tool in the authorization process. In effect it generates the whole sub tree for a given organization unit at runtime.

With this view in place, it is time to link it with the actual authorization process. We assume there is an (authenticated) *principal* identifier, who wants to perform an action on the *context* organization unit for which a role *code* is needed. Written as a function:

```
CREATE PROCEDURE [dbo].[AllowedToPerformAction]
  @principalId int,
  @contextId int,
  @roleCode varchar(50),
  @allowed int output
AS
BEGIN
  SELECT @allowed = COUNT(*) FROM
         OrganisationView INNER JOIN
  Role ON
         OrganisationView.PrincipalId = Role.PrincipalId AND
         OrganisationView.PathLevel >= Role.MinPathLevel AND
         OrganisationView.PathLevel <= Role.MaXPathLevel
  WHERE
         OrganisationView.PrincipalId = @principalId AND
         OrganisationView.ContextId = @contextId AND
         Role.code = @roleCode
END
```

The table below shows some results:

@principalId	@contextId	@roleCode	@allowed	Meaning
1 (CEO)	4 (Database administrator)	ModifyUserDetails	1	The CEO is allowed to modify user details of the database administrator
3 (Team manager)	6 (Junior software developer)	AssignTaskToUser	1	The team manager is allowed to assign tasks to the junior software developer
5 (Senior software developer)	6 (Junior software developer)	AssignTaskToUser	1	The Senior software developer is allowed to assign tasks to the junior software developer
5 (Senior software developer)	4 (Database administrator)	AssignTaskToUser	0	The Senior software developer is *not* allowed to assign tasks to the database administrator

Another utility stored procedure *CoverageSetOfRole* returns all contextual organization units that are included in this role for the given principal:

```
CoverageSetOfRole @principalId=3, @roleCode = 'AssignTaskToUser'
```

PrincipalId	PrincipalName	ContextId	ContextName	PathLevel
3	Team manager	3	Team manager	0
3	Team manager	4	Database administrator	1
3	Team manager	5	Senior software developer	1
3	Team manager	6	Junior software developer	2

The coverage set is very useful for user interfaces, where one wants to show an overview of the entities it can perform the actions on. It is good practice to only show those entities for which the actions are allowed. For example, when the team manager wants to assign tasks to a user, a dropdown list can be populated with the results of the coverage set. When there are multiple roles with the same role code assigned to a principal, then the coverage sets of these roles are *combined* to form a single coverage set for a role code for a principal.

Also note that the role table is *not* fully normalized, and for most systems it would make perfect sense to factor out some properties of the roles (role code, min path level, max path level) in a separate table and create an intermediate table that couples the organization units and roles. We have omitted this to make the model simpler and clearer.

This implementation fulfills the needs of our hierarchical role-based security framework. From this point on it should be fairly easy to implement authorization in a system by calling the correct stored procedures before executing functions that need authorization. Although fully functional, the implementation can be improved in terms of performance and functionality. In the next section we will explore these possibilities.

Improving the framework

The implementation that we have built so far is fully functional and works fine for smaller hierarchies. We found, however, that it does not scale very well when the hierarchy gets larger (10.000 organization units going four levels deep). One reason for this is the use of recursive CTE to traverse the tree. In our release of SQL server 2005 (including service pack 2), we found that with more complex queries the execution plan was sub-optimal and caused a lot of temporary intermediate lookup tables. Furthermore, we needed more information about the tree to implement *load-on-demand* in our user interface, which is a must-have if your tree gets larger. We have developed solutions to overcome these problems. These solutions will be presented in the following sections:

- Tree traversing in a more efficient way
- Automatic administration of depth of path and direct child count

Efficient tree traversing

Anyone that has been working with hierarchical structures in a relational database server knows it can become very cumbersome to work with. The core problem is the mismatch between the paradigm of database servers which is *set based* and the tree which is *hierarchy based*. There have been attempts at developing hierarchical database management systems, but none of these systems really hit the mainstream. An advantage of a relational DBMS is that it is very effective and efficient in answering queries that cross-cut over entities. There is a trade-off here between efficiently storing your hierarchical structure and efficiently querying a hierarchical structure.

In the setup we have implemented so far we have setup a naïve relational tree model (*Adjacency Lists*), which needs recursion to traverse. We will expand this model with features that enable us to use fast set-based operations on the tree. There are more ways to implement this. An article by Vadim Tropashko introduces different approaches to this[3]. We will focus on the *nested set* and *materialized path* approach here.

Nested set. We started out with implementing a *nested set*. In effect a nested set keeps track of the index of the node directly left and right of itself. The advantage of this approach is a very fast method of querying the tree. The disadvantage is that changes in the tree (updates and inserts) are very costly since the indices of roughly half of the nodes have to be recalculated. This can be a significant burden. In a tree with about 10.000 nodes every insert will roughly recalculate 5.000 nodes. In transactions these 5.000 node updates will be logged to provide rollback. With multiple insertions or updates the transaction log becomes huge and performance decreases unacceptable. There are more advanced solutions to these problems (by using real numbers instead of integers) but they are quite difficult to implement. If you intent on taking this course of action the article by Tropashko provides some pointers.

Materialized path. After the performance problems with the nested set approach we implemented a materialized path. In effect a materialized path defines the *global position* of a certain node within the tree. It defines the path going from the node up to the root node. Once this global position is calculated, it makes it easy to execute tree-wide queries since you can directly refer to this global

[3] http://www.dbazine.com/oracle/or-articles/tropashko4

position, instead of performing recursions for each node. The performance of this approach is very acceptable, and the costs of inserting or updating nodes are much less compared to the nested.

To implement the materialized path we define an extra field called `Path` in the table `OrganisationUnit`. This field will hold our materialized path. The field `PathLevel` will hold how deep the node is in the global tree.

Name	Type	Allows NULL
Id	int (primary key, identity increment)	No
ParentId	int	Yes
Name	varchar (100)	No
Path	varchar (200)	No
PathLevel	int	No

Furthermore, we want to ensure there is always a consistent and up to date path. A trigger linked to the `OrganisationUnit` table can do this.

```
CREATE TRIGGER [dbo].[UpdateMaterializedPath]
ON [dbo].[OrganisationUnit]
AFTER INSERT, UPDATE, DELETE
AS
IF UPDATE(ParentId) BEGIN
  WITH EmployeeLevels AS
  (
        SELECT
              Id,
              CONVERT(VARCHAR(MAX), Id) AS thePath,
              0 AS Level
        FROM OrganisationUnit
        WHERE ParentId IS NULL

        UNION ALL

        SELECT
              e.Id,
              x.thePath + '/' + CONVERT(VARCHAR(MAX), e.Id) AS thePath,
              x.Level + 1 AS Level
        FROM EmployeeLevels x
        JOIN OrganisationUnit e on e.ParentId = x.Id
  )
  UPDATE OrganisationUnit SET [OrganisationUnit].[Path]=thePath, PathLevel=Level FROM
        OrganisationUnit INNER JOIN EmployeeLevels ON  EmployeeLevels.Id =
OrganisationUnit.Id
  WHERE OrganisationUnit.Id IN (SELECT Id FROM inserted);
END
```

With this trigger in place our `OrganisationUnit` table will look like this:

Id	ParentId	Name	Path	PathLevel
1	*NULL*	CEO	1	0
2	1	Product manager	1/2	1
3	2	Team manager	1/2/3	2
4	3	Database administrator	1/2/3/4	3
5	3	Senior software developer	1/2/3/5	3
6	5	Junio software developer	1/2/3/5/6	4

With the global materialized path and path level present we can perform the following query to retrieve all descendants of an organization unit:

```
SELECT
  dbo.OrganisationUnit.Id,
  Child.Id AS ContextId,
  Child.PathLevel - dbo.OrganisationUnit.PathLevel AS PathLevel
FROM
  dbo.OrganisationUnit
  INNER JOIN
         dbo.OrganisationUnit AS Child
                ON Child.Path LIKE dbo.OrganisationUnit.Path + '%'
```

What does this query do? For each organization unit, all organization units are selected that start with the same path. There is no recursion involved here, and with an index on the `Path` field the query is very fast. The index can be utilized because the LIKE clause starts with a fixed path.

Because the roles can also work upwards in the tree we need a query that returns all ancestors of a given node. This is a bit more complicated, we could change the LIKE clause in the previous query as follows:

```
OrganisationUnit.Path LIKE Child.Path + '%'
```

But this will not perform well since no index can be used in this case and the whole table needs to be scanned. There is a good solution to this: we already know the ancestors, it is defined in the materialized path. All we need to do is extract the identifiers of the ancestors from the materialized path and look them up in the `OrganisationUnit` table.

For the extraction of the identifiers we define a user called `SplitIDs`:

```
CREATE FUNCTION [dbo].[SplitIDs]
(
  @IdList varchar(500),
  @Delimiter char(1) = '/'
)
```

```
RETURNS
@ParsedList table
(
  Id int
)
AS
  .. definition can be found in appendix A ..
END
```

Now you can use the new *CROSS APPLY* clause in MS SQL Server 2005 to execute this user defined function for each row. The result of the CROSS APPLY operation is a temporary table containing the identifiers of the ancestors. We can utilize this temporary table to inner join the ancestors.

```
SELECT  OrganisationUnit.Id, Parent.Id AS ContextId, Parent.PathLevel -
OrganisationUnit.PathLevel AS PathLevel
FROM     OrganisationUnit CROSS APPLY SplitIDs([OrganisationUnit].[Path], '/') AS
parentIds INNER JOIN
               OrganisationUnit AS Parent ON Parent.Id = parentIds.Id
```

The view that we utilize to map the tree to an intermediate table can now be defined as the union of the query that traverses upwards in the tree and the query that traverses downwards in the tree.

```
SELECT
  OrganisationUnit.Id,
  Parent.Id AS ContextId,
  Parent.PathLevel - OrganisationUnit.PathLevel AS PathLevel
FROM
  OrganisationUnit
         CROSS APPLY SplitIDs([OrganisationUnit].[Path], '/') AS parentIds
         INNER JOIN OrganisationUnit AS Parent
                 ON Parent.Id = parentIds.Id
UNION
SELECT
  dbo.OrganisationUnit.Id,
  Child.Id AS ContextId,
  Child.PathLevel - dbo.OrganisationUnit.PathLevel AS PathLevel
FROM
  dbo.OrganisationUnit
         INNER JOIN dbo.OrganisationUnit AS Child
                 ON Child.Path LIKE OrganisationUnit.Path + '%'
```

The stored procedures `AllowedToPerformAction` and `CoverageSetOfRole` make use of this view and are now automatically working with our new tree structure. There is no need to change these stored procedures.

The implementation with the materialized path has proven to be fast with test data up to 50.000 organization units. The cost of looking up coverage sets is O(log N) because of the index on the `Path` field. This means this implementation performs very well. Also we found that the generated execution plans are optimal even in more complex queries, in contrast with the implementation using recursive CTE which generated sometimes very poor execution plans resulting in poor performance.

Load-on-demand

For most applications it is worthwhile to invest in load-on-demand functionality in the presentation of you tree. This means that upon the first request you only show a subset of the tree (or coverage set) and you fetch other subsets of the tree only on demand of the client, for example when the client is expanding a node to see its children. To create load-on-demand we have create a second stored procedure, `CoverageSetOfRoleOnDemand`, to return the coverage set with two additional parameters:

- `@topOrganisationUnitId` – Defines the organisation unit id which is the root of the subset tree to return. The organisation unit itself is in the resultset.
- `@maximumDepth` – The number of levels to include in the resultset. A maximumDepth of 1 means only the direct children are in the resultset.

When a user expands an organization unit in the user interface, the system will fetch the underlying children that are covered by the role coverage set.

In the user interface you only want to show the option to expand an organization unit when it actually has children. Otherwise the user interface would just show no children, even though the user expected that expanding the organization unit would show its children. To solve this, and create a more usable user interface, we add an extra field to the `OrganisationUnit` table. This field, `ChildCount`, will hold the amount of direct children each organization unit has. Depending on `ChildCount > 0`, the user interface can show the option to expand or not.

To keep track of direct `ChildCount` we add a trigger to the `OrganisationUnit` table:

```
CREATE TRIGGER [dbo].[UpdateChildCount]
ON [dbo].[OrganisationUnit]
AFTER INSERT, UPDATE, DELETE
AS
IF UPDATE(ParentId) BEGIN
   UPDATE OrganisationUnit
   SET OrganisationUnit.ChildCount = OrganisationUnit.ChildCount -
         (SELECT COUNT(*) FROM deleted WHERE deleted.ParentId = OrganisationUnit.Id)
  WHERE OrganisationUnit.Id IN (SELECT ParentId FROM deleted)       ;

  UPDATE OrganisationUnit
  SET OrganisationUnit.ChildCount = OrganisationUnit.ChildCount +
         (SELECT COUNT(*) FROM inserted WHERE inserted.ParentId = OrganisationUnit.Id)
  WHERE OrganisationUnit.Id IN (SELECT ParentId FROM inserted);
END
```

This will automatically make sure `ChildCount` is always up-to-date.

Conclusion

We started with an introduction on the importance of security frameworks and the different approaches one can take in establishing a security framework. A common security framework makes use of role-based security whereby a principal is given a role which is valid system-wide. In many systems this is not powerful enough and needs to be more fine-grained and flexible. We developed a conceptual security framework based on *roles* combined with a *context*. The context is also part of a *hierarchy*. The roles we defined work either upwards or downwards in the hierarchy. This framework is both fine-grained and very flexible, which we showed with some examples of role usage. The conceptual framework defines basic data structures and operations. A basic but fully functional implementation was developed based on Microsoft SQL Server 2005. We identified two performance issues: first the use of recursion within the database and second the lack of load-on-demand. In the last section we extended the framework to resolve these issues. A full definition of the implementation is available in appendix A. The implementation can be used as-is or can be seen as a foundation from which a more elaborate security framework can be developed.

Appendix A Framework database definition

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[OrganisationUnit]') AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OrganisationUnit](
  [Id] [int] IDENTITY(1,1) NOT NULL,
  [ParentId] [int] NULL CONSTRAINT [DF_OrganisationUnit_ParentId]  DEFAULT (NULL),
  [Name] [varchar](100) NOT NULL,
  [Path] [varchar](200) NOT NULL CONSTRAINT [DF_OrganisationUnit_Path]  DEFAULT (''),
  [PathLevel] [int] NOT NULL CONSTRAINT [DF_OrganisationUnit_PathLevel]  DEFAULT ((0)),
  [ChildCount] [int] NOT NULL CONSTRAINT [DF_OrganisationUnit_ChildCount]  DEFAULT ((0)),
 CONSTRAINT [PK_OrganisationUnit] PRIMARY KEY CLUSTERED
(
  [Id] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
END
GO

IF NOT EXISTS (SELECT * FROM sys.indexes WHERE object_id =
OBJECT_ID(N'[dbo].[OrganisationUnit]') AND name = N'IX_OrganisationUnit')
CREATE UNIQUE NONCLUSTERED INDEX [IX_OrganisationUnit] ON [dbo].[OrganisationUnit]
(
  [Path] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
GO

IF NOT EXISTS (SELECT * FROM sys.indexes WHERE object_id =
OBJECT_ID(N'[dbo].[OrganisationUnit]') AND name = N'IX_OrganisationUnitParent')
CREATE NONCLUSTERED INDEX [IX_OrganisationUnitParent] ON [dbo].[OrganisationUnit]
(
  [ParentId] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO
```

```
IF NOT EXISTS (SELECT * FROM sys.triggers WHERE object_id =
OBJECT_ID(N'[dbo].[UpdateChildCount]'))
EXEC dbo.sp_executesql @statement = N'CREATE TRIGGER [dbo].[UpdateChildCount]
ON [dbo].[OrganisationUnit]
AFTER INSERT, UPDATE, DELETE
AS
-- Only update the childcount when ParentId field is changed
-- Note that the deleted and inserted table can contain multiple rows
IF UPDATE(ParentId) BEGIN
  -- Decrement the childcount with 1 for all parents of deleted nodes
  UPDATE OrganisationUnit
  SET OrganisationUnit.ChildCount = OrganisationUnit.ChildCount -
         (SELECT COUNT(*) FROM deleted WHERE deleted.ParentId = OrganisationUnit.Id)
  WHERE OrganisationUnit.Id IN (SELECT ParentId FROM deleted);

  -- Increment the childcount with 1 for all parents of inserted nodes
  UPDATE OrganisationUnit
  SET OrganisationUnit.ChildCount = OrganisationUnit.ChildCount +
         (SELECT COUNT(*) FROM inserted WHERE inserted.ParentId = OrganisationUnit.Id)
  WHERE OrganisationUnit.Id IN (SELECT ParentId FROM inserted);
END'
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

IF NOT EXISTS (SELECT * FROM sys.triggers WHERE object_id =
OBJECT_ID(N'[dbo].[UpdateMaterializedPath]'))
EXEC dbo.sp_executesql @statement = N'CREATE TRIGGER [dbo].[UpdateMaterializedPath]
ON [dbo].[OrganisationUnit]
AFTER INSERT, UPDATE, DELETE
AS
-- only update if the ParentId field has changed
IF UPDATE(ParentId) BEGIN
  -- calculates the path using recursive CTE
  WITH EmployeeLevels AS
  (
         SELECT
                Id,
                CONVERT(VARCHAR(MAX), Id) AS thePath,
                0 AS Level
         FROM OrganisationUnit
         WHERE ParentId IS NULL
```

```
        UNION ALL


        SELECT
                e.Id,
                x.thePath + ''/'' + CONVERT(VARCHAR(MAX), e.Id) AS thePath,
                x.Level + 1 AS Level
        FROM EmployeeLevels x
        JOIN OrganisationUnit e on e.ParentId = x.Id
  )
  -- assign the path to the new organisation unit
  UPDATE OrganisationUnit SET [OrganisationUnit].[Path]=thePath, PathLevel=Level FROM
        OrganisationUnit INNER JOIN EmployeeLevels ON  EmployeeLevels.Id =
OrganisationUnit.Id
  WHERE OrganisationUnit.Id IN (SELECT AffectedNode.Id FROM inserted INNER JOIN
OrganisationUnit as AffectedNode ON AffectedNode.Path LIKE inserted.Path + ''%'');
END









'
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[SplitIDs]') AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
BEGIN
execute dbo.sp_executesql @statement = N'CREATE FUNCTION [dbo].[SplitIDs]
(
  @IdList varchar(500),
  @Delimiter char(1) = ''/''
)
RETURNS
@ParsedList table
(
  Id int
)
AS
BEGIN
  DECLARE @Id varchar(10), @Pos int

  SET @IdList = LTRIM(RTRIM(@IdList))+ @Delimiter
  SET @Pos = CHARINDEX(@Delimiter, @IdList, 1)

  IF REPLACE(@IdList, @Delimiter, '''') <> ''''
```

```
  BEGIN
        WHILE @Pos > 0
        BEGIN
              SET @Id = LTRIM(RTRIM(LEFT(@IdList, @Pos - 1)))
              IF @Id <> ''''
              BEGIN
                    INSERT INTO @ParsedList (Id)
                    VALUES (CAST(@Id AS int)) --Use Appropriate conversion
              END
              SET @IdList = RIGHT(@IdList, LEN(@IdList) - @Pos)
              SET @Pos = CHARINDEX(@Delimiter, @IdList, 1)

        END
  END
  RETURN
END



'
END

GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Role]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Role](
  [PrincipalId] [int] NOT NULL,
  [Code] [varchar](50) NOT NULL,
  [MinPathLevel] [int] NOT NULL,
  [MaxPathLevel] [int] NOT NULL,
  [OrganisationUnitId] [int] NOT NULL
) ON [PRIMARY]
END
GO


IF NOT EXISTS (SELECT * FROM sys.indexes WHERE object_id = OBJECT_ID(N'[dbo].[Role]')
AND name = N'IX_RolePrincipalId')
CREATE NONCLUSTERED INDEX [IX_RolePrincipalId] ON [dbo].[Role]
(
  [PrincipalId] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
GO
SET ANSI_NULLS ON
```

```
GO

SET QUOTED_IDENTIFIER ON

GO

IF NOT EXISTS (SELECT * FROM sys.views WHERE object_id =
OBJECT_ID(N'[dbo].[OrganisationView]'))

EXEC dbo.sp_executesql @statement = N'CREATE VIEW [dbo].[OrganisationView]

AS

SELECT  OrganisationUnit.Id AS PrincipalId, Parent.Id AS ContextId, Parent.PathLevel -
OrganisationUnit.PathLevel AS PathLevel

FROM     OrganisationUnit CROSS APPLY SplitIDs([OrganisationUnit].[Path], ''/'') AS
parentIds INNER JOIN

               OrganisationUnit AS Parent ON Parent.Id = parentIds.Id

UNION

SELECT  dbo.OrganisationUnit.Id AS PrincipalId, Child.Id AS ContextId, Child.PathLevel -
dbo.OrganisationUnit.PathLevel AS PathLevel

FROM     dbo.OrganisationUnit INNER JOIN

               dbo.OrganisationUnit AS Child ON Child.Path LIKE OrganisationUnit.Path +
''%''

'

GO

SET ANSI_NULLS ON

GO

SET QUOTED_IDENTIFIER ON

GO

IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[CoverageSetOfRole]') AND type in (N'P', N'PC'))

BEGIN

EXEC dbo.sp_executesql @statement = N'CREATE PROCEDURE [dbo].[CoverageSetOfRole]

  @principalId int,

  @roleCode varchar(50)

AS

BEGIN

  SELECT Principal.Id as PrincipalId, Principal.Name as PrincipalName, Context.Id as
ContextId, Context.Name as ContextName, OrganisationView.PathLevel as PathLevel

    FROM

         OrganisationView INNER JOIN

  Role ON

         OrganisationView.PrincipalId = Role.PrincipalId AND

         OrganisationView.PathLevel >= Role.MinPathLevel AND

         OrganisationView.PathLevel <= Role.MaXPathLevel

  INNER JOIN

         OrganisationUnit as Principal ON

         OrganisationView.PrincipalId = Principal.Id

  INNER JOIN

         OrganisationUnit as Context ON

         OrganisationView.ContextId = Context.Id

  WHERE

         OrganisationView.PrincipalId = @principalId AND

         Role.code = @roleCode
```

```
END



'

END

GO

SET ANSI_NULLS ON

GO

SET QUOTED_IDENTIFIER ON

GO

IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[CoverageSetOfRoleOnDemand]') AND type in (N'P', N'PC'))

BEGIN

EXEC dbo.sp_executesql @statement = N'CREATE PROCEDURE [dbo].[CoverageSetOfRoleOnDemand]

  @principalId int,

  @roleCode varchar(50),

  @topOrganisationUnitId int,

  @maximumDepth int = 1

AS

-- Returns the subtree of nodes on which the function requiring the role identified by
@roleCode

-- is allowed to execute. Usefull for load-on-demand scenarios.

-- The subtree starts at the node having Id @topOrganisationUnitId and includes all its
descendants

-- that are also in the coverage set of the role.

-- When multiple roles are assigned to the principal the combined coverage set is
returned.

BEGIN

  SELECT Principal.Id as PrincipalId, Principal.Name as PrincipalName, Context.Id as
ContextId, Context.Name as ContextName, OrganisationView.PathLevel as PathLevel

    FROM

          OrganisationView INNER JOIN

  Role ON

          OrganisationView.PrincipalId = Role.PrincipalId AND

          OrganisationView.PathLevel >= Role.MinPathLevel AND

          OrganisationView.PathLevel <= Role.MaXPathLevel

  INNER JOIN

          OrganisationUnit as Principal ON

          OrganisationView.PrincipalId = Principal.Id

  INNER JOIN

          OrganisationUnit as Context ON

          OrganisationView.ContextId = Context.Id

  WHERE

          OrganisationView.PrincipalId = @principalId AND

          Role.code = @roleCode AND

          Context.Path LIKE (SELECT Path + ''%'' FROM OrganisationUnit WHERE
OrganisationUnit.Id=@topOrganisationUnitId) AND

          Context.PathLevel <= (SELECT PathLevel FROM OrganisationUnit WHERE
OrganisationUnit.Id=@topOrganisationUnitId) + @maximumDepth
```

```
END




'

END
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[AllowedToPerformFunction]') AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'CREATE PROCEDURE [dbo].[AllowedToPerformFunction]

  @principalId int,
  @contextId int,
  @roleCode varchar(50),
  @allowed int output
AS
-- returns whether the given principal @principalId is allowed to perform the
-- function requiring role identified by @rolecode on the context organisation
-- unit identified by @contextId
BEGIN
  SELECT @allowed=COUNT(*) FROM
         OrganisationView INNER JOIN
  Role ON
         OrganisationView.PrincipalId = Role.PrincipalId AND
         OrganisationView.PathLevel >= Role.MinPathLevel AND
         OrganisationView.PathLevel <= Role.MaXPathLevel
  WHERE
         OrganisationView.PrincipalId = @principalId AND
         OrganisationView.ContextId = @contextId AND
         Role.code = @roleCode
END




'

END
GO
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_OrganisationUnit_OrganisationUnit]') AND parent_object_id =
OBJECT_ID(N'[dbo].[OrganisationUnit]'))
```

```
ALTER TABLE [dbo].[OrganisationUnit]  WITH NOCHECK ADD  CONSTRAINT
[FK_OrganisationUnit_OrganisationUnit] FOREIGN KEY([ParentId])

REFERENCES [dbo].[OrganisationUnit] ([Id])

NOT FOR REPLICATION

GO

ALTER TABLE [dbo].[OrganisationUnit] NOCHECK CONSTRAINT
[FK_OrganisationUnit_OrganisationUnit]

GO

IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_Role_OrganisationUnitContext]') AND parent_object_id =
OBJECT_ID(N'[dbo].[Role]'))

ALTER TABLE [dbo].[Role]  WITH CHECK ADD  CONSTRAINT [FK_Role_OrganisationUnitContext]
FOREIGN KEY([OrganisationUnitId])

REFERENCES [dbo].[OrganisationUnit] ([Id])

GO

ALTER TABLE [dbo].[Role] CHECK CONSTRAINT [FK_Role_OrganisationUnitContext]

GO

IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_Role_OrganisationUnitPrincipal]') AND parent_object_id =
OBJECT_ID(N'[dbo].[Role]'))

ALTER TABLE [dbo].[Role]  WITH CHECK ADD  CONSTRAINT [FK_Role_OrganisationUnitPrincipal]
FOREIGN KEY([PrincipalId])

REFERENCES [dbo].[OrganisationUnit] ([Id])

GO

ALTER TABLE [dbo].[Role] CHECK CONSTRAINT [FK_Role_OrganisationUnitPrincipal]
```