Server based application level authorisation for Rotor

E. Rissanen

Abstract: Delegent is an authorisation server developed to provide a single centralised policy repository for multiple applications with support for decentralised administration by means of delegation. The author investigates how to integrate Delegent with the Rotor implementation of the .NET framework and compare the features of Delegent with those of the existing application level authorisation models of .NET. He concludes that Delegent offers help for application developers and a decentralised administration model, which are not available in standard .NET, and that the .NET model is well suited to be extended to use an authorisation server.

1 Introduction

Rotor [1] is a reference implementation of key parts of the .NET platform under a shared source licence, produced by Microsoft.

This paper is the result from research done with a grant from Microsoft Research in Cambridge. In our proposal we indicated that we have developed an authorisation server called Delegent [2], and we wish to study how we can integrate Delegent in the Rotor environment such that development of authorisation and its management in applications becomes easier and more secure.

2 .NET

In this Section we present a brief overview of some of the security features of .NET. We assume that the reader is familiar with the security model of .NET, and the presentation is very much biased to bring up only the issues that are relevant from the perspective of Delegent. Details can be found in [3] and [Note 1], for instance.

2.1 Code access security

Traditional operating system security is user identity based, that is, the identity of the user who is running a program will define what permissions the program has. The .NET model complements this with a code identity based model, where the identity of the code itself will define the permissions the program has. This code identity based model is called code access security.

Code access security in .NET is based on the concept of permissions. There are a number of permission classes, which represent permissions for actions such as reading files or opening windows.

The security policy for code access security is defined by identifying code in terms of attributes, such as digital signatures or file location, and then granting sets of available permissions to the code. The policy is stored in a number of configuration files. It is important to realise that code access permissions are granted to code, not to users. We deal with the trustworthiness of code, not users. It is not about whether James is allowed to read a file, but it is about whether we trust that the word processor James is using for reading the file is not a trojan.

Rotor/ .NET includes a run-time for so-called managed code, which is checked for type safety and memory accesses. By defining the permissions for components, and running them within the virtual machine, a sandbox can be created. This is how code access security is implemented.

The run-time, together with code in the system libraries, makes sure that code follows the security policy by checking that there are permissions for each attempted operation. Since, in the end, all operations must be performed by unmanaged code that calls to the operating system, there are a number of trusted components available. These components are granted the permission to call to unmanaged code. The components enforce security policy by demanding permissions. A permission demand throws an exception if the permission was not available to the requester.

A possible attack would be for malicious code to call code with special permissions, and to ask the code to do things that the malicious code itself cannot do. This is a so-called Luring Attack. To prevent it, the run-time performs a stack walk. When a permission is demanded, the run-time will check that each stack frame is associated with a component that has the demanded permission.

2.2 Role based security

The standard .NET distribution contains a role based security model. It is based on the PrincipalPermission class. PrincipalPermission looks like a code access permission but behaves differently. It is not for setting policy based on code identity, but on user identity. By demanding a PrincipalPermission it is possible to require that a specified

© IEE, 2003
IEE Proceedings online no. 20030991
doi: 10.1049/ip-sen:20030991
Paper received 29th June 2003

The author is with the Swedish Institute of Computer Science (SICS), Box 1263, 164 29 Kista, Sweden

Note 1: .NET Framework Documentation included with the .NET development kit from Microsoft.

user is running the code, and/or that the user who is running the code belongs to a specified role.

An executing thread may be associated with a principal object, which represents the user on whose behalf the code is running. The principal object may also contain information about which roles the user is a member of. The principal object is defined by the IPrincipal interface. One implementation, WindowsPrincipal, which is not implemented in Rotor, contains the windows log-on identity of a user and its windows groups as roles. There is a call available to set the current principal to the windows user that is running the code (also not implemented in Rotor). Another implementation, GenericPrincipal, can be used by applications that do their own authentication and role management. It contains the user identity and a list of strings that name the roles of the user.

Thus the role model of PrincipalPermission depends on which kind of principal objects the application is using. The Windows user groups form a simple role based access control model. Whether windows groups can be nested depends on whether the machine is stand alone or part of a domain. Groups in domains can be nested, with some limitations depending on the configuration of the domain. Since Rotor does not implement WindowsPrincipal, it falls somewhat outside the scope of this paper, so we are not going into the details of it.

The role model when using GenericPrincipal depends on the specific implementation of authentication and thus the initialisation of the role list. Also, the IsInRole() method of GenericPrincipal can be overridden, so any role model can be implemented by subclassing GenericPrincipal (or by implementing IPrincipal in a custom principal class all together).

2.3 Administration

How role administration is done depends on what kind of principal objects the application uses. In the case of WindowsPrincipals the administration is done with the regular Windows system administration tools. The tools provide for some form of delegation and decentralisation of administration, but none of this is available in Rotor. For GenericPrincipals the administration depends on the implementation of the specific authentication module or subclass of GenericPrincipal. There is no delegation or decentralisation of administration out of the box.

3 Delegent

Delegent is an authorisation server for application level user identity based authorisations that is developed at SICS. Delegent makes heavy use of delegation. The word ‘delegation’ has several meanings in the literature and in the commercial world. In this paper, there are two uses of the word. One is the organisational aspect of delegation, that is, assigning a duty to a subordinate. The Delegent server does not address this meaning of the word ‘delegation’. Instead, delegation in the scope of Delegent means the creation of new rights, which is the consequence of organisational delegation, since the subordinate will need the appropriate permissions to perform the delegated organisational duties.

Delegent uses an XML-based protocol for communication. This communication can optionally be protected by SSL and mutual authentication. There are some standardised protocols for authorisation systems, such as SAML [4], but they do not support all the features of Delegent, most notably policy updates and policy examination, in the way we want.

3.1 Motivation

The purpose of Delegent is to allow decentralised management of authorisations in a controlled and secure manner. Delegent shifts the focus from ‘who is authorised for an access?’, to ‘who is authorised to manage rights for an access?’ [5]. This can be applied recursively, such that authority hierarchies can be built, and administration over local matters can be delegated to local administrators.

The separation between administrative rights and access rights is very clear in Delegent, so it is possible, for instance, for a high level manager to have the right to decide on who may have a particular permission without having the permission himself. It is also possible to specify that he is not allowed to grant those permissions to himself.

The benefits from the delegation are several. Since authorisations can be distributed close to decision makers, the risks associated with a potentially corrupt central omnipotent administrative staff are reduced. The reduced distance may also make an organisation more efficient since the overhead with administration is reduced.

Delegent is implemented in the form of an authorisation server. An authorisation server provides authorisation related services such as storage of authorisation data, access control and management models, management of access control and auditing services. An authorisation server consolidates the authorisation related functions of one or more applications to one place, which saves development costs and provides a more clear and secure architecture. We believe authorisation servers will become a standard part of future IT-platforms.

3.2 Access control model

The access control model of Delegent is based on permissions in the form of user, object, method and time constraint tuples. There are no negative permissions in Delegent, and all access level permissions are in the form of such tuples. Any of the user, object and method fields may be a group. Table 1 shows some examples of access level permissions.

The user, object and method names are defined by application developers, and have no significance to Delegent. In this case we would mean that user John is permitted to withdraw money from the account during the years 2003 to 2004, and that Anne is permitted to deposit money on the account during the same period. Everyone in the group Auditors would be permitted to view all accounts.

Groups may include other groups, thus allowing inheritance of permissions.

In the Delegent model all authorisations are defined in terms of delegations that are issued by users. The access permissions (as above) are a special case. There is another kind of authorisation: administrative authorisations, which authorise the creation of other authorisations.

Each delegation has an issuer and a time stamp. Delegent will check that at the given time point the specified user had the authority to delegate the authorisation. If that is the case, the authorisation becomes active. In the case of an administrative authorisation, it will enable delegation of other authorisations, and in the case of an access level

Table 1

Subject	Object	Method	Time interval
John	Account 1234	Withdraw	2003-2004
Anne	Account 1234	Deposit	2003-2004
Auditors	All accounts	View	2003-2004

permission, it will become a part of the active access control matrix.

The way which delegations authorise other delegations creates chains of delegations, which may end in an access level permission. All valid access level permissions must be traced by such a chain to a source of authority, SoA, of that permission. The SoA of an authorisation is defined by means of a special delegation. Since in a delegation it is possible to constrain further delegation, for instance to limit delegation within a specified group of users, the SoA and intermediate delegators can get a centralised control over how an authorisation may propagate.

For details of the delegation model, we refer to [5–7].

3.3 Implementation

Delegent is implemented in the form of a networked server. The server will accept an access query in the form of a tuple of an atomic user, object and method, and will search its database of valid permissions. If a matching permission is found, a positive response is given. If there is no such permission, the response is negative.

3.4 Administration

Administration of permissions in Delegent is done by means of the delegation model of Delegent. Changes to the policy are done by either issuing a new Delegation or by revoking an existing one. The delegation model permits the decentralisation of administration of both access level permissions and administrative permissions.

4 Comparison

In this Section we present a comparison of the security in .NET with Delegent and explain how they complement each other.

The access control model of Delegent is comparable to the PrincipalPermission model in .NET. Both are used to define application/business level security policy on the basis of user identity and attributes. In contrast, the code access security model of .NET serves a different purpose, so it is not comparable to Delegent. However, it complements Delegent, as we will see below.

The PrincipalPermission framework gives no support for maintaining lists of which roles are required for which actions, so if an application uses the PrincipalPermission framework for its security policy, the application must either keep track of the security policy itself or it must be complemented with some kind of policy module. In either case it represents additional work for application developers. With Delegent, it is enough for the application developers to know the name of the action in the form of object and method names, since this is enough for Delegent to make an access control decision based on the contents of the policy repository. Also, the policy repository, which is stored within Delegent, can be shared between several applications if they operate on the same objects. We could, for instance, have one application for editing confidential documents and another viewing them. Both could use the same policy database, and request permissions on the same objects. In other words, in contrast to PrincipalPermission, Delegent offers a shared application independent policy repository out of the box.

There may be performance issues with Delegent based authorisation, since Delegent is a networked server. However, the purpose of Delegent is to provide authorisation on the 'business level', where demand may not be as high as at, for instance, file system level. Also there are many potential ways to increase performance in the future, for instance in the form of caching or partial reasoning on the application side.

Delegent offers a clear distinction between administrative rights and access rights. Again, whether PrincipalPermission offers this depends on the particular implementation of role administration.

5 An authorisation server based architecture

In this Section we present an overall architecture for how to decouple policy from applications and place it in a central policy repository. We wish to consolidate application level authorisation related functionality to Delegent, which will act as an authorisation service.

We assume that there is some kind of data or service that needs authorisation services. We call this the target. The authorisation services may be access control or auditing, for instance. The target is used by an application of some kind. We also need authentication of the users of the application, which is provided by a specific component. Finally there is Delegent, which provides the authorisation services.

5.1 Components

Below is a list of the components and their descriptions (see Fig. 1):

- *application*: acts on behalf of a user to perform some function on the target. In the case of a legacy application that cannot be modified, we may be able to use a proxy between the user and the application. In those cases, the proxy is analogous to the application in the discussion below
- *Delegent server*: provides the authorisation service and policy decision point. In this case we study only access control decisions, but other possible services include auditing and examination and management of security policy
- *target*: provides some kind of service or data that the application/user needs access to
- *enforcement function*: code that acts as the access point for use of the target. The purpose of the enforcement function is to enforce the security policy for accessing the target. The enforcement function will use Delegent for access control decisions
- *component that does authentication*: the application provides the credentials of the user to the authenticator, which will set the principal of the thread to the authenticated user. (In other cases the authenticator may obtain the user credentials directly from the user, and be located in between the user and the application.)

5.2 Authentication

There are a number of authentication requirements between components.

- The enforcer must authenticate itself to the application, so that the application can know it is dealing with the proper target. This also means that the application must trust the enforcer to pass on the request to the correct target.

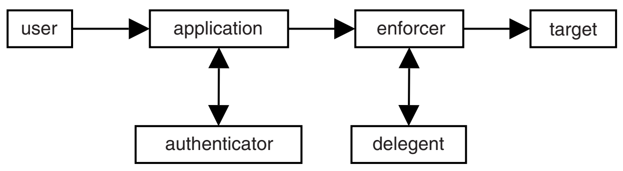

Fig. 1 *Components*

• The target must authenticate itself to the enforcer, so that the enforcer is not fooled by a bogus target.
• Delegent must authenticate itself to the enforcer, so that the enforcer is not fooled by an attacker's fake Delegent.
• We must make sure that all requests to the target pass the enforcer. To do this we may need to authenticate the enforcer to the target. An alternative to this is to use a custom code access permission which the target demands and is granted to only the legitimate enforcer. Yet another alternative is to use operating system level access control and physical security to prevent any use of the target outside the enforcer.
• The authenticator must authenticate itself to the application, so an attacker cannot lure the application to reveal the user's credentials.

In addition it may be desirable to restrict access to the enforcer/target with custom code access permissions to only trusted applications. This is to prevent trojans from performing access without the consent of the user.

6 Implementation

In this Section we present a sample implementation of the architecture described above. Not everything described above was implemented. In particular, authentication between components was not done and the communication channel to Delegent was left unencrypted.

6.1 DelegentPermission

Connecting Delegent to Rotor was simpler than expected. We wanted Delegent to follow the .NET 'look and feel', which required the creation of a new class called DelegentPermission that implements IPermission. We did not need to modify the run-time itself, which we to some degree had expected before we commenced work.

DelegentPermission is similar to PrincipalPermission. PrincipalPermission is not a code access security permission, so it does not perform a stack walk. A stack walk would be meaningless anyway, since the principal is associated with the thread, not the stack frames. For the same reason, DelegentPermission does not perform a stack walk.

A DelegentPermission contains a list of object/method pairs. Demanding a DelegentPermission will query Delegent over the network to see whether the principal associated with the current thread is allowed to all objects/methods in the list. If not, a SecurityException is raised.

Since the principal object of the current thread is used as the authentication token of the user it is important that the machine is configured such that the principal object cannot be set by untrusted code. There is a specific code access security permission for that in the form of the ControlPrincipal flag in SecurityPermission.

6.2 The application

As an example we have chosen a simple banking application. We have a number of Account objects that represent bank accounts. In this case they will act both as enforcers and targets. An account object will demand two permissions when an operation is attempted on it. First it will demand an AccountPermission, which is a custom code access permission. The purpose of this is to make sure only trusted applications can use the Account objects. It will also demand a Delegent permission. The purpose of this is to know that the user is authorised for the operation.

For the sample implementation we have a very simple authenticator component with a hardcoded user name and password. The application, AccountExample.exe, will ask for a user name and password on the console, and supply those to the Authenticator. If the credentials are correct, the authenticator sets the principal of the thread. Then AccountExample.exe will instantiate a number of Account objects, which it will operate on by the command of the user. The user enters commands on a simple command line, and the application will invoke the corresponding method on the specified Account object. The account object will use DelegentPermission to enforce access control.

This sample application is 'written from scratch', so there was no problem in using Delegent. However, an existing legacy application may not be a simple case. The reason for this is that many existing applications have application specific access control mechanisms built in. To use Delegent instead, the enforcing mechanism, which intrinsically must be tightly coupled with the target, must be modified to use Delegent for access control decision. An alternative is to use a proxy between the user and the application. In any case, a modular design, which permits easy modification of the way access control decisions are made, is a good strategy for new applications.

Here is the code for the Account class:

```
using System;
using Delegent;
using AccountPermission;
using System.Security;
[assembly:AllowPartiallyTrustedCallers]
namespace Account
{
  public class Account
    {
    private double balance;
    private string number;
    private DelegentConnection
      connection;
    public Account(string number,
      double balance)
    {
      // tomat is the host where we run
         // Delegent
      connection = new DelegentConnection
        ("tomat", 4712);
      this.number = number;
      this.balance = balance;
    }
    public double Balance
    {
      get
      {
        return balance;
      }
    }
    public string Number
    {
      get
      {
        return number;
      }
    }
    public void withdraw (double amount)
    {
      AccountPermission.
        AccountPermission accperm =
        new AccountPermission.AccountPer-
          mission
        (number, AccountPermission.
          AccessType.Withdraw);
    accperm.Demand ();
    DelegentPermission    request = new
      DelegentPermission
      (connection, "account_" + number,
        "withdraw");
    request.Demand ();
    balance− = amount;
  }
  public void deposit (double amount)
  {
    AccountPermission.AccountPermis-
      sion accperm =
      new AccountPermission.
        AccountPermission
        (number, AccountPermission.
          AccessType.Deposit);
    accperm.Demand ();
    DelegentPermission    request = new
      DelegentPermission
        (connection, "account_" + number,
          "deposit");
    request.Demand ();
    balance+= amount;
  }
 }
}
```

The components are located in separate assemblies so we can assign separate security policies to them. The security policies of the various components become:

• *Authenticator:* This is granted permission for Assertion, Execution and ControlPrincipal. It needs these permissions to set the principal when called from partially trusted components.
• *DelegentPermission:* This is granted permission for Assertion, Execution and unrestricted SocketPermission and DnsPermission. These permissions are needed so DelegentPermission can make the call to Delegent over the network.
• *Account:* This is granted permission to execute. In this example it does not need any other permissions, since the balance is simply stored in a member of the object.
• *AccountExample.exe:* This is granted permission to execute and an AccountPermission. The AccountPermission represents that we trust this executable to handle accounts properly on behalf of the user.

The Account objects will perform an operation only if the operation was invoked by an authorised user from a trusted application. The application does not need to perform any authorisation calculations itself, and contains only user interface related code.

The security policy represents what kind of trust we put on the various components:

• Authenticator is trusted to verify user credentials and to set the principal of a thread accordingly.
• Account is trusted to allow only authenticated, authorised users to work on accounts and with only trusted applications.
• AccountExample.exe is trusted to be honest to the user when she uses it for accessing an account.

6.3 Administration

The above example illustrates only the use of access level permissions. We chose to do so, since the administration of the permissions does not directly relate to the integration of Delegent with the .NET security model. Administration would be done in an administration tool or with administrative functions in the banking application. The administration tools would use the special policy update and examination commands of the Delegent protocol.

7 Conclusion

.NET security has two complementary sides to it. One is the Code Access Security model, which is about trusting code. The other is user identity based security, which is about which users are permitted access. Delegent is an authorisation server for user identity based access control, so the access control model of Delegent is comparable to the PrincipalPermission model of .NET.

Delegent offers a decentralised administration model and a shared policy repository out of the box, which is not the case for PrincipalPermission. PrincipalPermission has a simple role based access control model in it, but how the roles are administered depends on what kind of principals are used and how the administration is implemented for the particular case. For WindowsPrincipals there are standard tools to decentralise administration. For custom principals, everything has to be custom made.

Code access security complements Delegent. It provides a means to establish a trusted path from the user to the target and to Delegent.

We believe that there is much to gain by using an authorisation server for application/business level authorisations, and our work shows that the .NET architecture is well suited for this. It is easy to extend the .NET security to use a server, and the existing security of .NET complements the approach. Benefits from an authorisation server include a centralised repository, decentralised and consolidated management and simpler application development.

8 Acknowledgments

The author wishes to thank Babak Sadighi for his advice and help in reviewing the paper.

9 References

1 http://msdn.microsoft.com/net/sscli/, accessed June 2003
2 http://delegent.com, accessed June 2003
3 '.NET framework security' (Addison Wesley, Boston, 2002)
4 http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security, accessed August 2003
5 Firozabadi, B.S., Sergot, M., and Bandmann, O.: 'Using authority certificates to create management structures'. Proc. 9th Int. Workshop, on Security Protocols, Cambridge, UK, April 2001, pp. 134-145
6 Firozabadi, B.S., and Sergot, M.: 'Revocation schemes for delegated authorities'. Proc. 3rd IEEE Int. Workshop on Policies for Distributed Systems and Networks, Monterey, CA, 5–7 June 2002, pp. 210–213
7 Bandmann, O., Dam, M., and Firozabadi, B.S.: 'Constrained Delegations'. Proc. IEEE Symposium on Security and Privacy, Berkeley, CA, 12–15 May 2002, pp. 131–190