Administration

For the administration we don’t need to user more complicated pattern than Identity Provider.
It’s simple and well secured for ours purposes. We don’t need to create a lot of services to control it, just one service. Simple and Fast.

4.3 Identity Provider

The IDENTITY PROVIDER pattern allows the centralization of the administration of subjects' identity information for a security domain.

Example

Having applied the CIRCLE OF TRUST pattern, we can trust the identity providers who are members of our federation, but we still have a variety of identities.

Context

One or several resources, such as web services, CORBA services, applications and so on that are accessed by a predetermined set of subjects. The subjects and resources are typically from the same organization.

Problem

Each application or service may implement its own code for managing subjects' identity information, leading to an overloading of implementation and maintenance costs that may lead to inconsistencies across the organization's units.

Solution

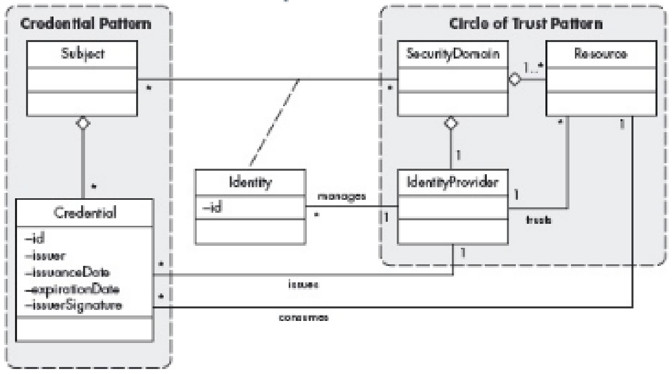
The management of the subjects' information for an organization is centralized in an IDENTITY PROVIDER, which is responsible for storing and propagating parts of the subjects' information (that form their identity) to the applications and services that need it.

We define a security domain as the set of resources whose subjects' identities are managed by the IDENTITY PROVIDER. Typically, the IDENTITY PROVIDER issues a set of credentials to each subject that will be verified by the accessed resources. Notice that the security domain is a special kind of CIRCLE OF TRUST within an organization.

Structure

Figure 4.3 shows a UML class diagram describing the structure of the solution. This pattern combines the CIRCLE OF TRUST, making explicit its Resources and specializing ServiceProvider to the IdentityProvider. Subjects are identified using CREDENTIALs given by a specific identity provider.

Figure 4.3: Class diagram for the IDENTITY PROVIDER pattern



Example Resolved

Now that we can trust the identity providers who are members of our federation and have a centralized identity managed by a specific identity provider, we do not need multiple identities.

Consequences

The IDENTITY PROVIDER pattern offers the following benefits:

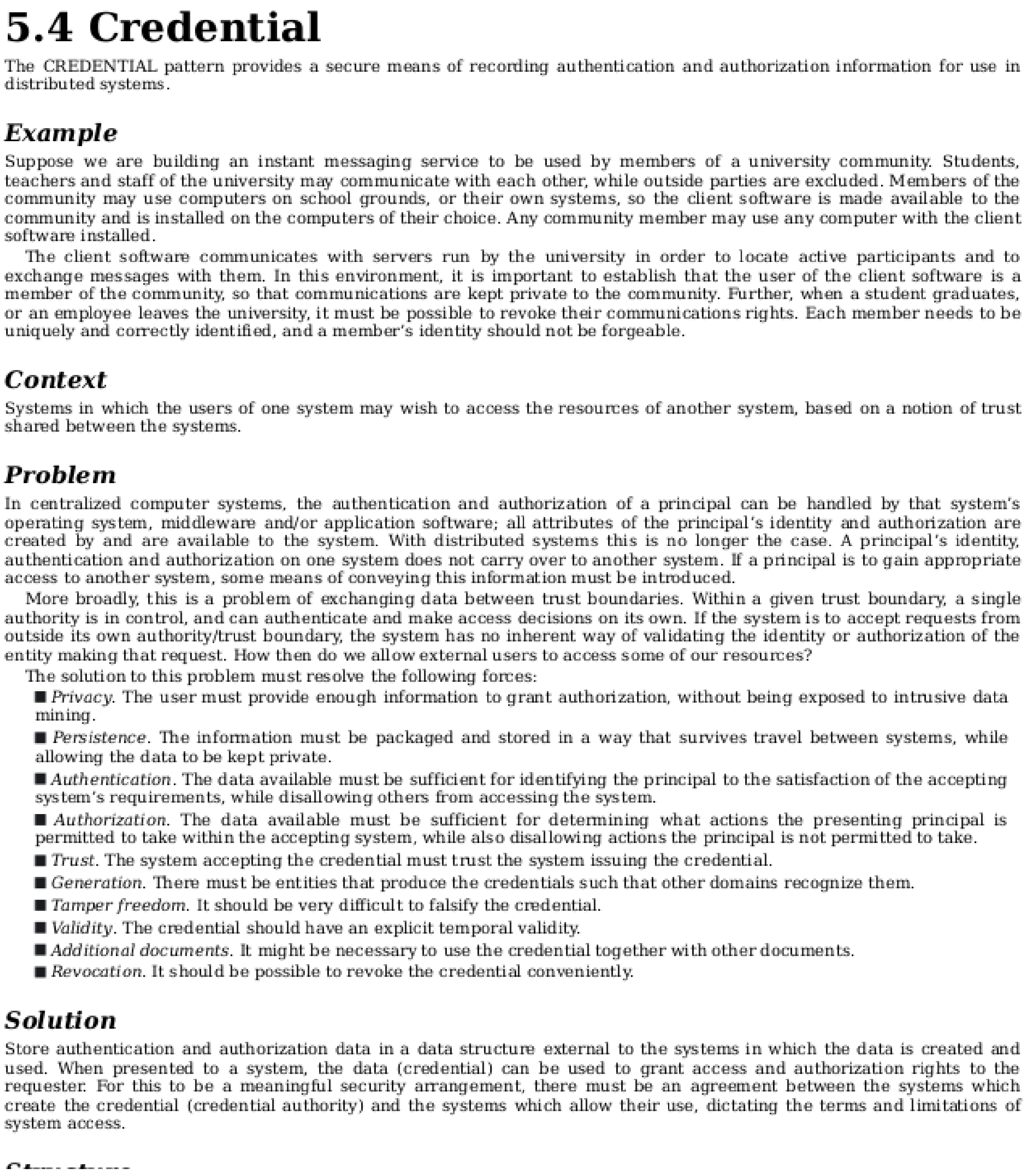
- Maintenance costs are reduced
- The system is consistent in terms of its users

It also suffers from the following liability:

- The IDENTITY PROVIDER can be a bottleneck in the organization's network.

Authentication

For authentication it’s better to use Credential pattern for us. Why? It’s a great choice if we wanted to create a well-distributed system with a lot of systems(micro-services) which can be managed from the main hub by users without any additional control system.



5.4 Credential

The CREDENTIAL pattern provides a secure means of recording authentication and authorization information for use in distributed systems.

Example

Suppose we are building an instant messaging service to be used by members of a university community. Students, teachers and staff of the university may communicate with each other, while outside parties are excluded. Members of the community may use computers on school grounds, or their own systems, so the client software is made available to the community and is installed on the computers of their choice. Any community member may use any computer with the client software installed.

The client software communicates with servers run by the university in order to locate active participants and to exchange messages with them. In this environment, it is important to establish that the user of the client software is a member of the community, so that communications are kept private to the community. Further, when a student graduates, or an employee leaves the university, it must be possible to revoke their communications rights. Each member needs to be uniquely and correctly identified, and a member’s identity should not be forgeable.

Context

Systems in which the users of one system may wish to access the resources of another system, based on a notion of trust shared between the systems.

Problem

In centralized computer systems, the authentication and authorization of a principal can be handled by that system’s operating system, middleware and/or application software; all attributes of the principal’s identity and authorization are created by and are available to the system. With distributed systems this is no longer the case. A principal’s identity, authentication and authorization on one system does not carry over to another system. If a principal is to gain appropriate access to another system, some means of conveying this information must be introduced.

More broadly, this is a problem of exchanging data between trust boundaries. Within a given trust boundary, a single authority is in control, and can authenticate and make access decisions on its own. If the system is to accept requests from outside its own authority/trust boundary, the system has no inherent way of validating the identity or authorization of the entity making that request. How then do we allow external users to access some of our resources?

The solution to this problem must resolve the following forces:

- *Privacy*. The user must provide enough information to grant authorization, without being exposed to intrusive data mining.
- *Persistence*. The information must be packaged and stored in a way that survives travel between systems, while allowing the data to be kept private.
- *Authentication*. The data available must be sufficient for identifying the principal to the satisfaction of the accepting system’s requirements, while disallowing others from accessing the system.
- *Authorization*. The data available must be sufficient for determining what actions the presenting principal is permitted to take within the accepting system, while also disallowing actions the principal is not permitted to take.
- *Trust*. The system accepting the credential must trust the system issuing the credential.
- *Generation*. There must be entities that produce the credentials such that other domains recognize them.
- *Tamper freedom*. It should be very difficult to falsify the credential.
- *Validity*. The credential should have an explicit temporal validity.
- *Additional documents*. It might be necessary to use the credential together with other documents.
- *Revocation*. It should be possible to revoke the credential conveniently.

Solution

Store authentication and authorization data in a data structure external to the systems in which the data is created and used. When presented to a system, the data (credential) can be used to grant access and authorization rights to the requester. For this to be a meaningful security arrangement, there must be an agreement between the systems which create the credential (credential authority) and the systems which allow their use, dictating the terms and limitations of system access.

Access control

For the access control I would like to introduce “Session-Based Role-Basses Access Control” pattern which is great for separating roles in the system. With it we can delegate certain rights to certain users. For example, if you are simple user you don’t need to have an access to settings of sensors, you just need to have rights to turn it on or off. And if you are an admin, you can change the delay or intensity of light, or temperature of water

6.10 Session-Based Role-Based Access Control

The SESSION-BASED ROLE-BASED ACCESS CONTROL pattern allows access to resources based on the role of the subject, and limits the rights that can be applied at a given time based on the roles defined by the access session.

Example

John is a developer on a project. He is also a project leader for another project. As a project leader he can evaluate the performance of the members of his project. He combines his two roles and adds several flattering evaluations about himself in the project where he is a developer. Later, his manager, thinking that the comments came from the project leader of the project on which John is a developer, gives John a big bonus.

Context

Any environment in which we need to control access to computing resources, in which users can be classified according to their jobs or their tasks, and in which we assign rights to the roles needed to perform those tasks.

We assume the existence of a Session pattern that can be used for the solution.

Problem

In an organization a user may play several roles. However, for each access the user must act only within the authorizations of a single role (that is, within the context of the role) or combinations of roles that do not violate institution policies. How can we force subjects to follow the policies of the institution when using their roles?

In addition to the forces defined for the CONTROLLED ACCESS SESSION pattern, the solution to this problem must resolve the following forces:

- People in institutions have different needs for access to information, according to their functions. They may have several roles associated with specific functions or tasks.
- We want to help the institution to define precise access rights for its members so that the least privilege policy can be applied when they perform specific tasks.
- Users may have more than one role and we may want to enforce policies such as *separation of duty*, where a user cannot be in two or more specific roles in the same session.

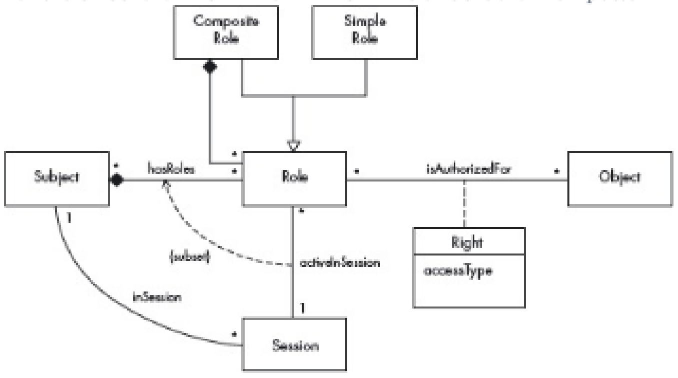
Solution

A subject may have several roles. Each role collects the rights that a user can activate at a given moment (execution context), while a session controls the way in which roles are used, and can enforce role exclusion at execution time.

Structure

The structure of the SESSION-BASED ROLE-BASED ACCESS CONTROL pattern is shown in the class diagram in Figure 6.18. The class Role is an intermediary between Subject and Object, holding all authorizations a user possesses while performing the role, and acts here as an execution context. Within a Session, only a subset of the roles assigned to a Subject may be activated, just those necessary to perform the intended task. Roles may be composed according to a Composite pattern [Gam94], in which higher-level roles acquire (inherit) rights from lower-level roles.

Figure 6.18: Class diagram for the SESSION-BASED ROLE-BASED ACCESS CONTROL pattern



Secure Management

With this pattern we can prevent attacks from the outside. It’s like “Firewall”. It’s will use more system and hardware resources but no one from the outside will get privileges to control our system.

7.6 Protected Entry Points

The PROTECTED ENTRY POINTS pattern describes how to force a call from one process to another to go through only prespecified entry points where the correctness of the call is checked and other access restrictions may be applied.

Example

ChronOS is a company building a new operating system, including a variety of plug-in services such as media players, browsers and others. In their design, processes can call each other in unrestricted ways. This makes process calls fast, which results in generally good performance, and everybody is satisfied. However, when they test the system, an error anywhere produces problems, because it propagates to other processes, corrupting their execution. Also, many security attacks are shown to be possible. It is clear that when their systems are in use they will acquire a bad reputation and ChronOS will have problems selling it. They need to have a system that provides resilient service in the presence of errors, and which is resistant to attacks.

Context

Executing processes in a computing system. Processes need to call other processes to ask for services or to collaborate in the computation of an algorithm, and usually share data and other resources. The environment can be centralized or distributed. Some processes may be malicious or contain errors.

Problem

Process communication has an effect on security, because if a process calls another using entry points without appropriate checks, the calling process may read or modify data illegally, alter the code of the executing process, or take over its privilege level. If the checks are applied at specific entry points, some languages, such as C or C++, let the user manipulate pointers to bypass those entry points. Process communication also has a major effect on reliability, because an error in a process may propagate to others and disrupt their execution.

The solution to this problem must resolve the following forces:

- Executing processes need to call each other to perform their functions. For example, in operating systems user processes need to call kernel processes to perform I/O, communications and other system functions. In all environments, process may collaborate to solve a common problem, and this collaboration requires communication. All this means that we cannot use process isolation to solve this problem.
- A call must go to a specified entry point or checks could be bypassed. Some languages let users alter entry point addresses, allowing input checks to be bypassed.
- A process typically provides services to other processes, but not all services are available to all processes. A call to a service not authorized to a process can be a security threat or allow error propagation.
- In a computing environment we have a variety of processes with different levels of trust. Some are processes that we normally trust, such as kernel processes; others may include operating system utilities, user processes and processes of uncertain origin. Some of these processes may have errors or be malicious. All calls need to be checked.
- The number, type and size of the passed parameters in a call can be used to attack a process, for example by producing a buffer overflow. Incorrect parameters may produce or propagate an error.

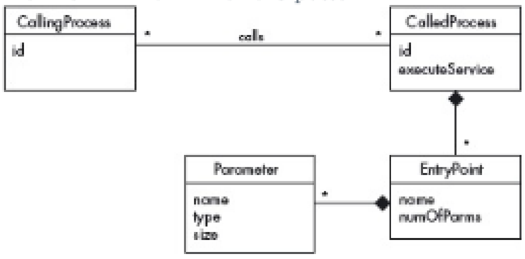
Solution

Systems that use explicit message passing have the possibility of checking each message to see if it complies with system policies. For example, one security feature that can be applied when calling another process is *protected entry points*. A process calling another process can only enter the called process at predesigned entry points, and only if the signature used is correct (name, number of parameters, type and size of parameters). This prevents bypassing entry checks and avoids attacks such a buffer overflows.

Structure

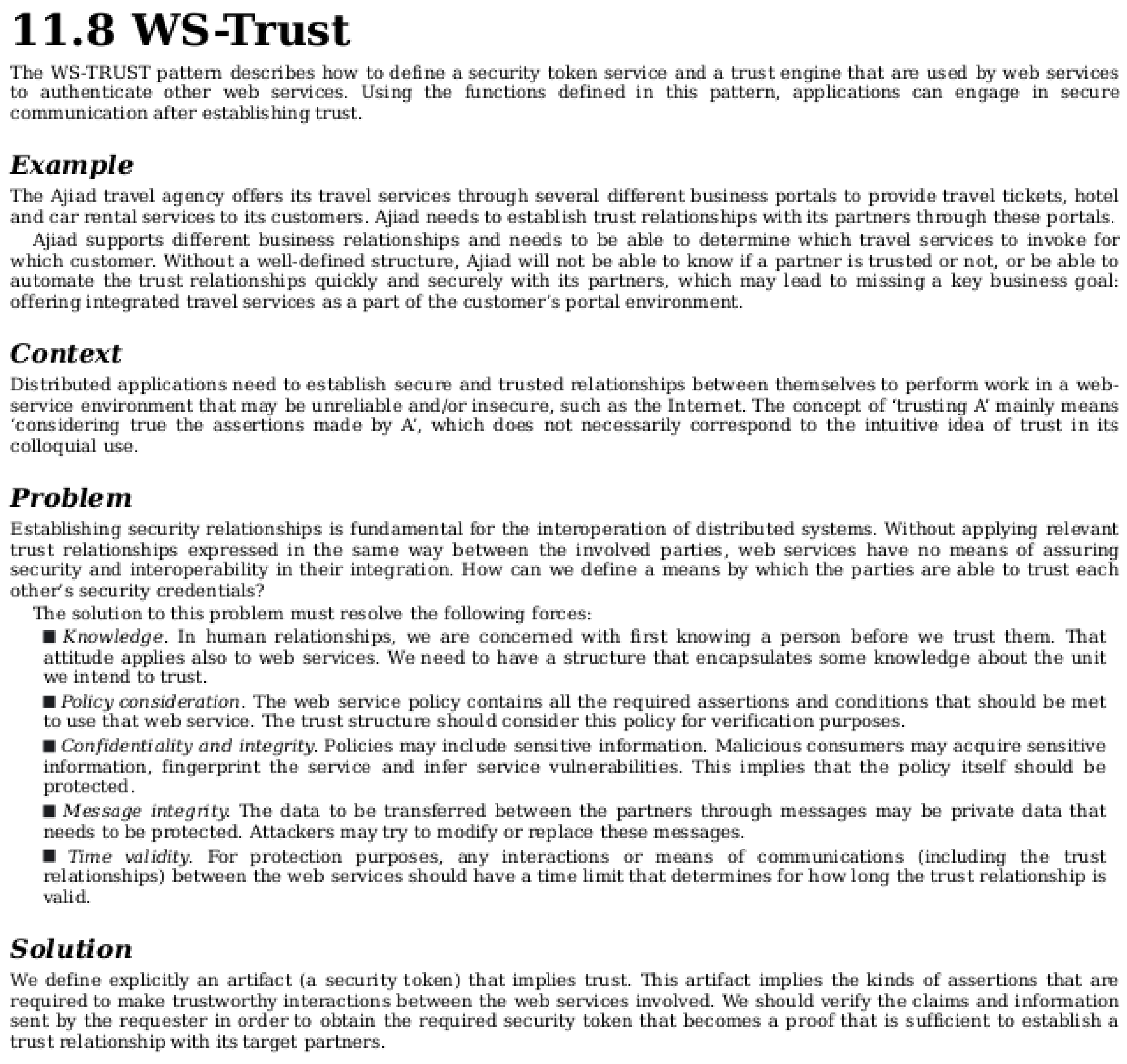
Figure 7.11 shows the class diagram of the solution. CallingProcess and CalledProcess are roles of processes in general. When a CallingProcess makes a request for a service to another process, the request is handled by an EntryPoint. This EntryPoint has a name and a list of parameters with predefined numbers, types, and size limits that can be used to check the correctness of the call signature. It can optionally add access control checks by using a Reference Monitor pattern or other input data tests.

Figure 7.11: Class diagram for the PROTECTED ENTRY POINTS pattern



Web Services Security

We can use the WS-Trust pattern to establish a secure connection between our system and other programs that can obtain control of the system. The additional token service will check if this program is in the list of programs that are adapted and certified for our system or not.



11.8 WS-Trust

The WS-TRUST pattern describes how to define a security token service and a trust engine that are used by web services to authenticate other web services. Using the functions defined in this pattern, applications can engage in secure communication after establishing trust.

Example

The Ajiad travel agency offers its travel services through several different business portals to provide travel tickets, hotel and car rental services to its customers. Ajiad needs to establish trust relationships with its partners through these portals.

Ajiad supports different business relationships and needs to be able to determine which travel services to invoke for which customer. Without a well-defined structure, Ajiad will not be able to know if a partner is trusted or not, or be able to automate the trust relationships quickly and securely with its partners, which may lead to missing a key business goal: offering integrated travel services as a part of the customer's portal environment.

Context

Distributed applications need to establish secure and trusted relationships between themselves to perform work in a web-service environment that may be unreliable and/or insecure, such as the Internet. The concept of 'trusting A' mainly means 'considering true the assertions made by A', which does not necessarily correspond to the intuitive idea of trust in its colloquial use.

Problem

Establishing security relationships is fundamental for the interoperation of distributed systems. Without applying relevant trust relationships expressed in the same way between the involved parties, web services have no means of assuring security and interoperability in their integration. How can we define a means by which the parties are able to trust each other's security credentials?

The solution to this problem must resolve the following forces:

- *Knowledge*. In human relationships, we are concerned with first knowing a person before we trust them. That attitude applies also to web services. We need to have a structure that encapsulates some knowledge about the unit we intend to trust.
- *Policy consideration*. The web service policy contains all the required assertions and conditions that should be met to use that web service. The trust structure should consider this policy for verification purposes.
- *Confidentiality and integrity*. Policies may include sensitive information. Malicious consumers may acquire sensitive information, fingerprint the service and infer service vulnerabilities. This implies that the policy itself should be protected.
- *Message integrity*. The data to be transferred between the partners through messages may be private data that needs to be protected. Attackers may try to modify or replace these messages.
- *Time validity*. For protection purposes, any interactions or means of communications (including the trust relationships) between the web services should have a time limit that determines for how long the trust relationship is valid.

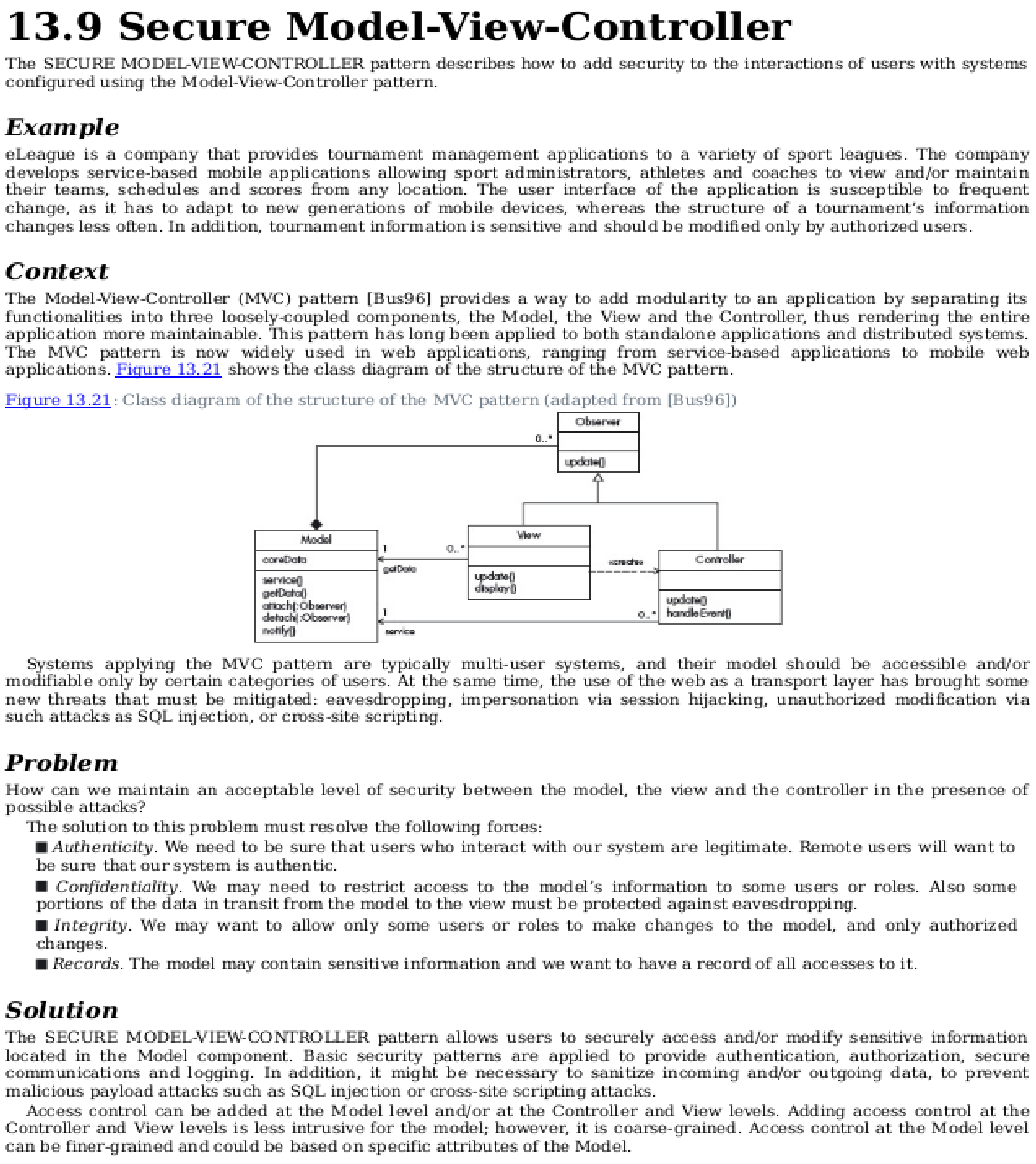
Solution

We define explicitly an artifact (a security token) that implies trust. This artifact implies the kinds of assertions that are required to make trustworthy interactions between the web services involved. We should verify the claims and information sent by the requester in order to obtain the required security token that becomes a proof that is sufficient to establish a trust relationship with its target partners.

Secure Middleware

Here we have two interesting patterns which are very similar.
1) Secure Three-Tier Architecture
2) Secure Model-View-Controller

They are almost the same in theory but different in implementation. They divide the app in 3 layers to delegate the responsibility but I think more preferable will be the Secure Model-View-Controller (MVC) pattern because it's well known design of application and easy extendable for our security needs.



13.9 Secure Model-View-Controller

The SECURE MODEL-VIEW-CONTROLLER pattern describes how to add security to the interactions of users with systems configured using the Model-View-Controller pattern.

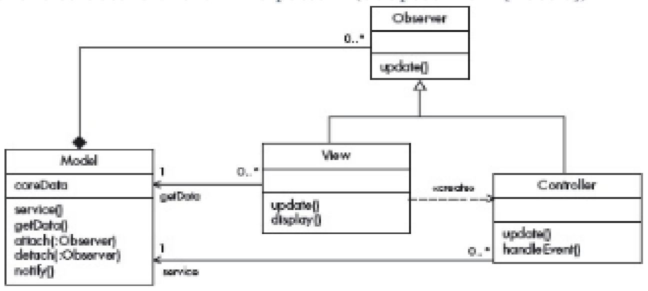
Example

eLeague is a company that provides tournament management applications to a variety of sport leagues. The company develops service-based mobile applications allowing sport administrators, athletes and coaches to view and/or maintain their teams, schedules and scores from any location. The user interface of the application is susceptible to frequent change, as it has to adapt to new generations of mobile devices, whereas the structure of a tournament's information changes less often. In addition, tournament information is sensitive and should be modified only by authorized users.

Context

The Model-View-Controller (MVC) pattern [Bus96] provides a way to add modularity to an application by separating its functionalities into three loosely-coupled components, the Model, the View and the Controller, thus rendering the entire application more maintainable. This pattern has long been applied to both standalone applications and distributed systems. The MVC pattern is now widely used in web applications, ranging from service-based applications to mobile web applications. Figure 13.21 shows the class diagram of the structure of the MVC pattern.

Figure 13.21: Class diagram of the structure of the MVC pattern (adapted from [Bus96])



Systems applying the MVC pattern are typically multi-user systems, and their model should be accessible and/or modifiable only by certain categories of users. At the same time, the use of the web as a transport layer has brought some new threats that must be mitigated: eavesdropping, impersonation via session hijacking, unauthorized modification via such attacks as SQL injection, or cross-site scripting.

Problem

How can we maintain an acceptable level of security between the model, the view and the controller in the presence of possible attacks?

The solution to this problem must resolve the following forces:

- *Authenticity*. We need to be sure that users who interact with our system are legitimate. Remote users will want to be sure that our system is authentic.
- *Confidentiality*. We may need to restrict access to the model's information to some users or roles. Also some portions of the data in transit from the model to the view must be protected against eavesdropping.
- *Integrity*. We may want to allow only some users or roles to make changes to the model, and only authorized changes.
- *Records*. The model may contain sensitive information and we want to have a record of all accesses to it.

Solution

The SECURE MODEL-VIEW-CONTROLLER pattern allows users to securely access and/or modify sensitive information located in the Model component. Basic security patterns are applied to provide authentication, authorization, secure communications and logging. In addition, it might be necessary to sanitize incoming and/or outgoing data, to prevent malicious payload attacks such as SQL injection or cross-site scripting attacks.

Access control can be added at the Model level and/or at the Controller and View levels. Adding access control at the Controller and View levels is less intrusive for the model; however, it is coarse-grained. Access control at the Model level can be finer-grained and could be based on specific attributes of the Model.