Model-based Run-time Checking of Security Permissions using Guarded Objects[⋆]

Jan Jürjens[1⋆⋆]

Computing Department, The Open University, GB

Abstract. In this paper we deal with the application of run-time checking to enforce requirements which, because of their nature, cannot be enforced statically. More specifically, it deals with the problem how to control access to objects within an object-oriented system at run-time in a way that enforces an overall security policy. It aims to improve on the ad-hoc (and often untrustworthy) way it is currently done in practice by automatically generating the run-time checks from a model-based specification of the system that captures the security policy. Concretely, the models are expressed in the UML security extension UMLsec, and the run-time checks that are generated for Java programs rely on GuardedObjects.

1 Introduction

Since IT systems become more and more interconnected, they also become exposed to an increasing number of attacks. In order to develop high quality systems, it is therefore important to consider security aspects in the software development process. Security is a complex non-functional requirement which can only be guaranteed by the interaction of many parts in a system. Leaving security aspects to late stages and not considering them systematically makes their integration extremely difficult and increases the potential for the final product to contain vulnerabilities.

A commonly used security concept is permission-based access control, i.e. associating entities (e.g., users or objects) in a system with permissions and allowing an entity to perform a certain action on another entity only if it owns the necessary permissions. Designing and enforcing a correct permission-based access control policy (with respect to the general security requirements) is very hard, especially because of the complex interplay between the system entities. This is aggravated by the fact that permissions can also be delegated to other objects for actions to be performed on the delegating object's behalf.

Especially dynamic access control mechanisms such as provided by Java since the JDK 1.2 security architecture [Gon99] in the form of GuardedObjects can be

⋆ This work was partially funded by the Royal Society within the project Model-based Formal Security Analysis of Crypto Protocol Implementations.

⋆⋆ `http://www.jurjens.de/jan`

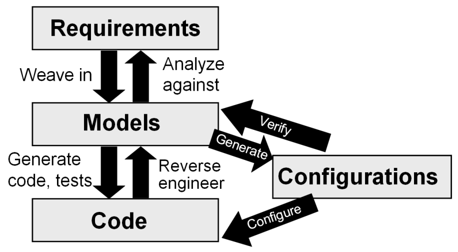

Fig. 1. Model-based Security Engineering

difficult to administer since it is easy to forget an access check. If the appropriate access controls are not performed, the security of the entire system may be compromised. Additionally, access control may be granted indirectly and unintentionally by granting access to an object containing the signature key that enables access to another object.

In this paper, we present an approach for the integration of run-time checks to enforce permissions into early design models, in particular for object-oriented design using UML. We both describe static modelling aspects, where we introduce owned and required permissions and capabilities for their delegation into class diagrams, and dynamic modelling aspects. Dynamic modelling aspects are characterized by the use and delegation of permissions within a interaction of the system objects, modelled as a sequence diagram. To gain confidence in the correctness of the permission-based access control policy, we define checks for the consistency of the permission-related aspects within the static and dynamic models and between these models. For the implementation as run-time checks, we show a way to transfer permissions using cryptographic certificates and provide a formal analysis. We also address the realization of the run-time checks in Java using GuardedObjects. We demonstrate our approach at the example of a model of an instant message service.

The work presented here is part of a more general approach towards model-based security engineering visualized in Fig. 1 (see [Jür04]).

In the next section, we will give the necessary background on permission-based security in object-oriented systems. In Sect. 3, the above described aspects will be modelled in UML. Sect. 4 addresses consistency checks of the UML model, whereas Sect. 5 deals with security aspects of the model in an abstract way. The integration of the modelled concepts into a concrete programming language like Java is described in Sect. 6.

2 Run-time checks for security permissions in OO systems

Objects in object-oriented systems usually interact with each other in the following way: one object becomes an actor and performs an action on another (passive) object. The passive object will be changed or activated by these actions. Activation means that the entity will also become an actor in order to perform actions on other entities.

In security-critical systems, it is crucial to have control over the execution of the actions. For this purpose, the execution of the actions is controlled by permissions. An actor is only allowed to initiate actions on certain objects when he owns the associated permissions.

In the context of such a model, we denote objects which own or define permissions as *permission-secured* objects. Permission-secured objects are the smallest entities on which permissions can be defined. Not every object in a system must be a permission-secured object. The permissions are attached to the actions that can be performed on an object. It is possible to define several permissions, which must all be owned for performing an action. In the following, we write the names of objects, classes and methods in *italics*, and denote permissions and corresponding model annotations in sansserif.

As an example, we consider a simple file. There is the permission-secured object *file*. The file defines two protected actions: *read*, which is protected by the permission read, and *write*, protected by the permission write and the permission read. These permissions are valid for the whole *file* object. We assume the protection of lower levels like lines or characters is not possible.

There are two types of owning permissions: there are permissions which are defined statically, but there is also the possibility to delegate permissions to other objects. Delegation is necessary to enable an activated object to fulfil the jobs an actor has given to it, but where the activated object itself does not have the necessary rights.

In this case, it should be possible for the delegate to act in the name of the actor. For this purpose, it is necessary to restrict the given permissions to the ones actually needed, to limit security risks. It is also necessary that it is always recognizable that the delegate acts in commission. Therefore re-delegation of permissions to other objects is an important issue. It is possible that the delegating object does not know the final delegate at delegation time.

An example for a re-delegation is the use of an account statement printer. The account owner wants to get his account statement and initiates the process. As he is not able to enquire the banking host system himself, he charges the account statement printer with doing this. Therefore, he gives the permission for reading the account information to the machine, in order for it to get the information in behalf of the account owner.

As we regard this example in more detail, it turns out that it is suitable to restrict this authorization because of two aspects:

– The printer should be able to make use of the authorization only once. If it is possible to use the authorization more than once, the printer can print the owners account balance to every other customer.
– There should be a timeout, after which the authorization expires. If there is no timeout, it is possible that the printer makes use of the authorization after the customer has left the bank.

3 Specifying run-time checks for security permissions using UMLsec

To model the permission-based security aspects of a system, we must identify the permission-secured objects. The smallest entities on which actions may be executed are the objects. Thus, every object that is defined in a system may be a permission-secured object.

The next step is to define the protected actions. The usual way to change an object's state from the outside is to invoke a corresponding method. So it is necessary to treat method invocations as security-critical actions and thus to protect methods by permissions.

Another way of changing an object's state is to read or write public attributes directly. Although public attributes are not a good way of object design, reading and writing them must also be regarded as an action. While it is possible to restrict access to public variables at the modelling level of a system, it is not possible to do so in common object oriented programming languages. The best way to cope with this problem is to use only private variables in combination with *get* and *set* methods.

Let us reconsider the file system example. In this case, we can model two objects: a *file* object and a *line* object, where the *file* object is an aggregation of many *line* objects. The only way to access the *line* objects is to use the methods of the corresponding *file* object. The methods of the *file* object are protected by permissions, whereas the methods the *line* object places at the *file* object's disposal are not. As these methods are only available to the *file* object, it is not necessary to make the *line* object a permission-secured object.

In a first step, we will look at the static description of the permissions in an system model. After that, we describe the permission-related aspects of the dynamic interaction of the system regarding the activities the system is designed for.

3.1 Static definitions

First, we describe the static aspects of an integration of permission-based security into UML models. For this purpose, we consider class diagrams and deal with the following questions:

- Which classes define permission-secured objects?
- Which permissions will be assigned to these objects at instantiation time? These permissions are the same for all objects instantiated from the same class.
- Which methods (and public attributes) will be protected by permissions?
- What kinds of permissions are these?
- Which of the assigned permissions may be delegated? How can one define to which type of objects they may be delegated?

First of all, the permission-secured objects will be identified by marking classes that define or own permission objects with the stereotype «permission−secured». If an object owns certain permissions on other objects at instantiation time, this is also stated at this place. A tagged value is associated with the «permission − secured» stereotype consisting of a list of tuples structured as follows: $\{\mathsf{permission} = [(\mathit{class}, \mathsf{permission})]\}$. The first parameter of the tuple indicates the class on which the permission is valid. The second parameter names the permission.

Methods and public attributes to which access is restricted are marked with the stereotype «permissioncheck» and an associated tagged value containing the list of permissions needed for access ($\{\mathsf{permission} = [\mathsf{permission}]\}$). This list is only a simple list naming the permissions. The association to classes is given by the class implementing the method or containing the attribute. To allow objects of certain classes classified as reliable unrestricted access to particular methods and public variables, it is possible to associate a second tag to the stereotype «permission_check». The tagged value $\{\mathsf{no\_permission\_needed} = [\mathit{class}]\}$ indicates that objects of the named classes need no permissions for access.

Although delegation is a dynamic process, which comes into effect at execution time, at this point of view it is of interest which permissions can be delegated at all, and if so, to which class of objects these permissions may be delegated.

Classes that can delegate at least some of their permissions have the following tag: $\{\mathit{delegation} = [(\mathit{class}, \mathit{permission}, \mathit{role}/\mathit{class})]\}$. The first two parameters name the permission which is delegated together with the class it belongs to. The third parameter names the class to which the permission can be delegated.

The last aspect to be regarded in the static class definition is inheritance. Definitions belonging to the modelling of permissions are inherited in the same way as all other definitions are inherited. Redefining a method or an attribute makes it necessary to also redefine the stereotypes and tags for permission modelling.

Now let us describe the example of the Instant Messaging Service which will be used to illustrate the definitions in the remainder of this paper. The class diagram in Fig. 2 shows the *SubscriptionClient* and the *InstantMessenger* on the client side, which define permission-secured objects. The class *SubscriptionClient* contains the permission subscribe on objects of class *SubscriptionServer* and the permission receive on objects of class *InstantMessenger*. In the model, this is reflected by the tagged value {permission=[(*SubscriptionServer*, subscribe), (*InstantMessenger*, receive)]}. The latter permission is marked for delegation to objects of class *Forwarder*. This is defined by the tag $\{\mathit{delegation} = [(\mathit{InstantMessenger}, \mathit{receive}, [\mathit{Forwarder}])\}$.

On the server side, there are the classes *SubscriptionServer* and *Forwarder*. The class *SubscriptionServer* gets the permission forward on objects of class *Forwarder*. The access to the method *subscribe()* is guarded by the permission subscribe. This is stated by the stereotype «permission_check» and the tag $\{\mathsf{permission} = [\mathsf{subscribe}]\}$. For calling the method *checkLogin()*, the possession of

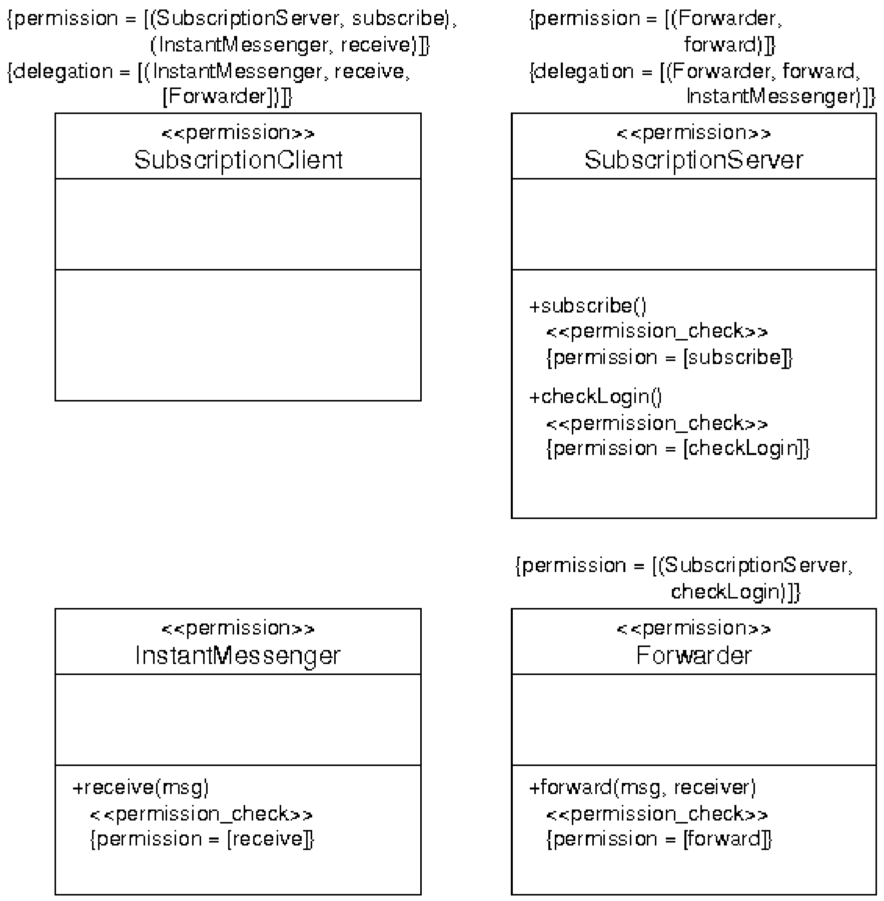
{permission = [(SubscriptionServer, subscribe), (InstantMessenger, receive)]}
{delegation = [(InstantMessenger, receive, [Forwarder])]}
<<permission>>
SubscriptionClient
{permission = [(Forwarder, forward)]}
{delegation = [(Forwarder, forward, InstantMessenger)]}
<<permission>>
SubscriptionServer
+subscribe()
<<permission_check>>
{permission = [subscribe]}
+checkLogin()
<<permission_check>>
{permission = [checkLogin]}
{permission = [(SubscriptionServer, checkLogin)]}
<<permission>>
InstantMessenger
+receive(msg)
<<permission_check>>
{permission = [receive]}
<<permission>>
Forwarder
+forward(msg, receiver)
<<permission_check>>
{permission = [forward]}

Fig. 2. Class diagram for instant messager

the permission checkLogin is necessary. The class *Forwarder* defines the method *forward(msg, receiver)*, which is guarded by the permission forward.

3.2 Dynamic definitions

In this section, the point of view on the system changes to the modelling of interactions between the objects instantiated from the classes defined above. For that purpose, we identify and model workflows.

For workflow modelling in UML, activity diagrams are used, where activities are assigned to the objects. It is possible to depict the interaction of several objects solving one problem regarding the causal and temporal dependencies.

For the description of used and needed permissions, we often need more detailed information than activity diagrams can offer. The main problem arises from the possibility to combine a number of single actions into one activity, which is connected with a number of objects. For coping with permissions in an automated way, one needs to identify the objects communicating with each other clearly. This means that for every single action we must be able to name the sender and the receiver to coordinate the necessary permissions. This information easily gets lost when aggregating actions to activities. For this reason it is only possible to use activity diagrams to catch the workflow whereas for further use the workflow must be converted to a sequence diagram. In a sequence diagram one can identify caller and callee in every single step of communication, which allows to assign the permissions to the sent messages.

For refinement of the workflow, a sequence diagram is created, allowing to specify the connection between permissions and messages by regarding the exchange of messages between objects. In a first step, we define which of the objects are permission-secured objects, using the same stereotype $\ll \mathsf{permission} - \mathsf{secured} \gg$ as in the class diagram. To this stereotype, we attach the permissions the object owns on other objects, utilizing tagged values. These tags are defined the same way as in the class diagrams, by $\{\mathsf{permission} = [(object, \mathsf{permission})]\}$. In contrast to the class diagram, here the first parameter of the tuple means no longer a class but a concrete object on which the permission is valid. Additionally, the ability for delegation of certain permissions is stated by a tag as well ($\{\mathsf{delegation} = [(\mathsf{class}, \mathsf{permission}, \mathsf{role/class})]\}$).

Permissions which are needed for executing a method – or in other words for sending a message successfully – are attached directly to the message which is to be protected by these permissions. To signalize that a message is protected by permissions, the message is marked with the stereotype $\ll \mathsf{permission\_check} \gg$, where the permissions are named as tagged values ($\{\mathsf{permission} = [\mathsf{permission}]\}$).

The delegation is performed by emitting and passing on certificates, which are formally defined as 7-tuples

$$\mathsf{certificate} = (e, d, c, o, p, x, s)$$

with emittent e, delegate d, class c of the delegate, object o, permission p which is valid on o, expiration timestamp x and sequence number s.

A certificate contains the following information:

– Who is delegating a permission? The emittent e is named in the certificate; he is signing the certificate.
– To whom is the permission to be delegated? For the definition of the delegate, there are two possibilities, depending on the relation between emittent and delegate. If the emittent knows the delegate at emission time of the certificate, he can name him explicitly (field d in the certificate). Otherwise, he can name the class c the delegate must be an instance of to make use of this certificate. In this case, d has the value *null.* In our example, the emittent never knows the delegate, thus the latter (more general) type of certificate is used.

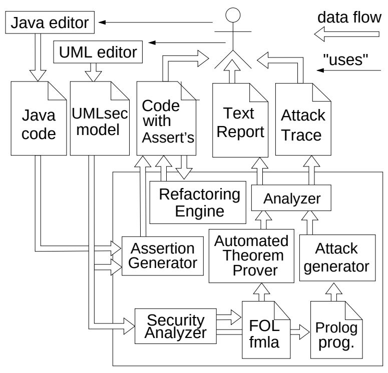

Fig. 3. Model-based Security Tool Suite

- Which permission is to be delegated? The permission to be delegated is defined by two parameters: the permission p and the object o on which this permission is valid.
- For how long is the permission to be delegated? As it is not possible to define a contiguous time in sequence diagrams, it is also not possible to make temporal restrictions on the validity of certificates. Time will be approximated by the number of messages to be sent, starting at zero with the first message. Thus, if a certificate is valid unrestrictedly, this parameter is set to -1.
- What about the sequence number? The sequence number s is contained in the certificate to avoid that it is used several times. The sequence number of certificates which are defined by the same parameter values must differ. It is also necessary that the number is the same if a certificate is passed along several objects. For defining a certificate which might be used more than once, this parameter is to be set to -1.

In the sequence diagram, messages where permission certificates are sent are marked by the stereotype «certification», where a 7-tuple representing a certificate will be directly attached as an tagged value. The parameters of this tag correspond to the definition above.

4 Statically verifying the run-time checks specified in UMLsec

In this section, we explain how one can statically verify the run-time checks that can be specified on the UMLsec model level (as explained in the previous section) against security requirements. This is supported by the security checkers provided within the UMLsec tool suite [Too08, JY07, JS08] (Fig. 3).

4.1 Consistency between class and sequence diagrams

As class diagrams and sequence diagrams are linked very closely to each other regarding the security permissions, it is necessary to check the consistency of the definitions made in these two diagrams.

In the class diagram, classes are assigned permissions on other classes. The definitions made there have to correspond to the definitions of the objects instantiated out of these class definitions. This means that objects must not have been assigned definitions, which are not contained in the corresponding class definition. It is only admissible to define less permissions in the sequence diagram than in the class diagram.

The definitions for delegation are treated in a similar way, with some restrictions. In the sequence diagram, only permissions can be delegated for which this possibility is defined in the class diagram. Besides that, it is necessary that the permission which is to be delegated is present, which means that it is not only defined in the class definition, but also in the object definition.

The next thing to check is the definition of methods. The permissions needed to execute a single method are defined in the class diagram. It is necessary that these definitions fit the definitions of the sequence diagrams. The method calls are defined as messages there. Attached to these messages are the permissions which are necessary to force the receiver to execute the message in the desired way. Therefore, it is necessary that these permissions are consistent with the ones defined in the receiver's class definition.

4.2 Dynamic checking of the sequence diagram

Are all permissions assigned in a system in a way that the processes modeled in the sequence diagram are able to be completed? This is the next question to solve. If an object should be able to send a message, it must own all permissions necessary for that action. Permissions which are assigned statically are not a problem (addressed by the consistency checks described above), but permissions assigned dynamically by delegation are:

– A permission certificate must be received before it can be used, which means both using the permission included in the certificate and passing on the certificate to other objects.
– The emittent of a certificate must be able to create the certificate. This means that he must own the permission statically and the permission must be released for delegation.
– A certificate must be valid at time of use. The loss of validity will be defined by a time stamp in the certificate.
– A certificate which is defined for being used only once looses validity by being used, so no object can use it again.

In the sequence diagram for the instant messaging service the object *Sender* calls the method *forward()* of *ForS* where the permission forward is needed. As the object *Sender* does not own this permission, it is delegated by an certificate which is passed on by the message *create()*:

$$\{certificate = (SubS, null, ForS, forward, InstantMessenger, -1, -1)\}$$

Because of lack of this permission, *SubSender*, the sender of this message, cannot create this certificate must receive it from *SubS* by sending *subscriptionConfirmation(). SubS* owns the permission and is able to delegate it. One can see this by the tags assigned to this object:

- $\{\mathsf{permission} = [(\mathsf{ForS}, \mathsf{forward})]\}$
- $\{\mathsf{delegate} = [(\mathsf{ForS}, \mathsf{forward}, \mathsf{InstantMessenger})]\}$

The period of validity has not been considered in this example, because no time stamp is available. Also, the certificates may be used more than once.

5 Run-time checks for permission delegation

A permission is a message consisting of *permission* and *identifier* (of the object the permission is valid on). The object owning the permission will be specified by appending the object's public key. Therefore it is impossible for any other object to use this permission. A *certificate* is defined as a triple consisting of the identifier followed by the permission and the public key of the user of the *certificate.*

For signing the permissions, there is a trusted instance in the system called security authority (SA). This instance releases all permissions and passes them on to the objects at their instantiation time. It is not possible to change the definition of a permission once signed by this authority.

So the a certificate defining a permission will be formally defined as follows: $Sign(identifier::permission::K_{legitimate}, K_{SA}{}^{-1})$.

To enable the delegation of permissions, passing on the permission is not enough. The delegating object must issue a certificate containing the permission and restrictions for its use. In addition, the certificate contains the public key of the owner of the permission. This allows other objects to prove that this object originally was the owner of the permission. The certificate is be signed with the private key of the permission's owner:

$$Sign(K_{legitimate} :: Sign(object :: permission :: K_{legitimate}, K_{SA}^{-1}) :: [properties], K_{legitimate}^{-1})$$

Making use of a delegated permission is only allowed for objects which are implementing the properties of the properties-list.

Here, we now have to deal with the usual Dolev-Yao attacker model:

- The intruder can save all messages sent between objects.
- Messages can be deleted by the intruder, so that the receiver is not able to get a specific message
- The intruder is able to insert messages into the communication between objects

Table 1. Notation for cryptographic expressions

inv(k)	Inverse key of *k*; a message, encrypted with key *k* can be decrypted by *inv(k)*.
sign(E,inv(k))	The message E is signed with the inverse key *inv(k)*.
enc(E,k)	The message E is encrypted with the key *k*.
conc(E1,E2)	A message consists of two concatenated single messages E1 and E2.
fst(E)	Inversion of conc(E1,E2); gives back the first element E1 of the concatenation.
snd(E)	Inversion of conc(E1,E2); gives back the second element E1 of the concatenation.
ext(E, k)	Extracts the message E out of a message signed message with the inverse key *inv(k)* of *k*.
dec(E, inv(k))	Decrypts the message E out of a message encrypted message with the inverse key *inv(k)* of *k*.

– By combination of these threats, the intruder is able to manipulate messages.

As usual, one makes use of cryptography to try to avoid such attacks by encrypting messages. In the case of security permissions it must be ensured that only the legitimate object is able to make use of a permission. Although by the definition of permissions it is guaranteed that only legitimate objects are able to create certificates for granting permissions, it is possible for intruders to obtain such a certificate in order to use the included permission. This threat can only be avoided by using an additional encryption mechanism for transmitting these certificates.

For proving such a modelling we enhance the UML model by cryptographic functions given in Table 1 for producing a protocol for secure communication between the objects following [Jür04]. The security check for this protocol is done automatically using the first-order predicate logic automated theorem prover e-Setheo. For this, the protocol is converted into predicates in the TPTP-syntax following the formal semantics for UML given in [Jür04].

We explain the modelling of such a protocol by the example of the instant messaging service. For simplification, only the communication between sender and server will be regarded. In the communication with the receiver, it is assumed that the *Forwarder ForS* obtained the permission receive on the receiver-object before using it.

In Figure 4, the corresponding sequence diagram is shown. The notation for cryptographic expressions used in this diagram is given in Table 1. For better readability, the messages contain names of functions (such as subscribe or conf), indicating their purpose. On the receiving side, the components of the received messages are referred to by A_1, A_2 and A_3 (parameters of the subscribe message), respectively by B_i, C_i, and D_i (parameters of the messages conf, init and forward). Note that the protocol in Figure 4 is only considered as an example to demonstrate our approach, not necessarily as an optimal solution for the situation at hand.

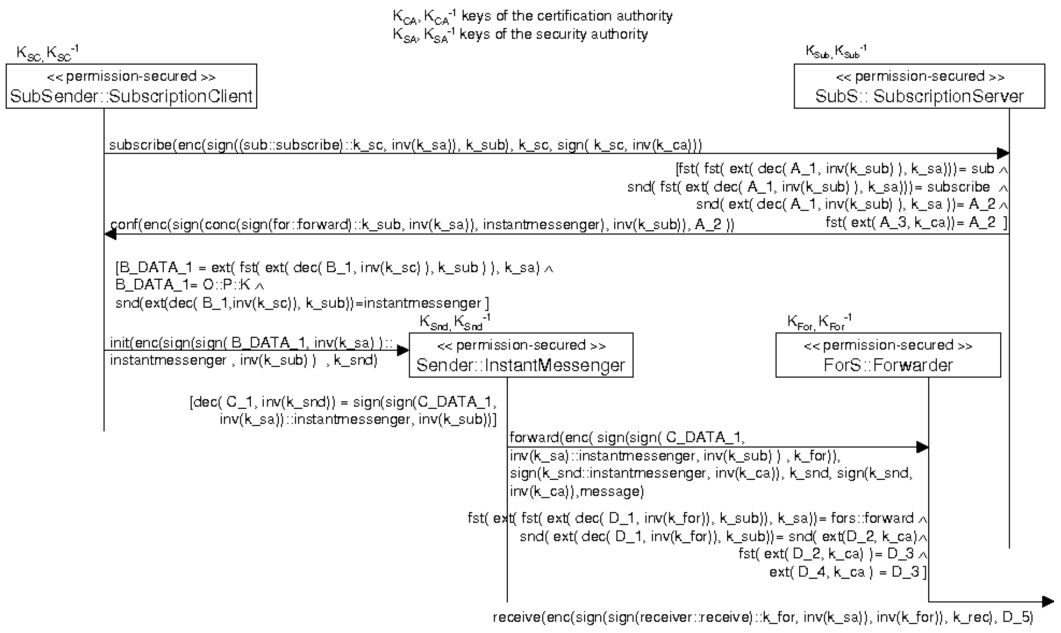
K_{CA}, K_{CA}^{-1} keys of the certification authority
K_{SA}, K_{SA}^{-1} keys of the security authority
K_{SC}, K_{SC}^{-1}
<< permission-secured >>
SubSender::SubscriptionClient
K_{Sub}, K_{Sub}^{-1}
<< permission-secured >>
SubS:: SubscriptionServer
subscribe(enc(sign((sub::subscribe)::k_sc, inv(k_sa)), k_sub), k_sc, sign(k_sc, inv(k_ca)))
[fst(fst(ext(dec(A_1, inv(k_sub)), k_sa)))= sub ∧
snd(fst(ext(dec(A_1, inv(k_sub)), k_sa)))= subscribe ∧
snd(ext(dec(A_1, inv(k_sub)), k_sa))= A_2 ∧
fst(ext(A_3, k_ca))= A_2]
conf(enc(sign(conc(sign(for::forward)::k_sub, inv(k_sa)), instantmessenger), inv(k_sub)), A_2))
[B_DATA_1 = ext(fst(ext(dec(B_1, inv(k_sc)), k_sub)), k_sa) ∧
B_DATA_1= O::P::K ∧
snd(ext(dec(B_1,inv(k_sc)), k_sub))=instantmessenger]
K_{Snd}, K_{Snd}^{-1}
<< permission-secured >>
Sender::InstantMessenger
K_{For}, K_{For}^{-1}
<< permission-secured >>
ForS::Forwarder
init(enc(sign(sign(B_DATA_1, inv(k_sa))::
instantmessenger , inv(k_sub)) , k_snd)
[dec(C_1, inv(k_snd)) = sign(sign(C_DATA_1,
inv(k_sa))::instantmessenger, inv(k_sub))]
forward(enc(sign(sign(C_DATA_1,
inv(k_sa)::instantmessenger, inv(k_sub)) , k_for)),
sign(k_snd::instantmessenger, inv(k_ca)), k_snd, sign(k_snd,
inv(k_ca)),message)
fst(ext(fst(ext(dec(D_1, inv(k_for)), k_sub)), k_sa))= fors::forward ∧
snd(ext(dec(D_1, inv(k_for)), k_sub))= snd(ext(D_2, k_ca)∧
fst(ext(D_2, k_ca))= D_3 ∧
ext(D_4, k_ca) = D_3]
receive(enc(sign(sign(receiver::receive)::k_for, inv(k_sa)), inv(k_for)), k_rec), D_5)

Fig. 4. Delegation protocol

As specified in Figure 4, the object *SubSender* connects to the server *SubS* and delivers the necessary certificate *sign(conc(conc(SubS,subscribe),*K_{SC}*), inv(*K_{SA}*))* to the Server, which was signed by the security authority with key *inv(*K_{SA}*)*. It is encrypted with the public key K_{Sub} of *SubS* to ensure that only *SubS* can access the message. When *SubS* gets the message, it checks the permission and the certification of the public key. If the check is successful, an acknowledgement is sent back to *SubSender* that contains a permission certificate allowing an object of class *InstantMessenger* to send messages to the Forwarder *ForS*. This certificate consists of the following parameters:

– the permission, signed by the security authority,
– the name of the class *InstantMessenger*, so that only objects of that class are able to use the permission,

For a secure transmission, the certificate is encrypted with the public key K_{SC} of *SubSender*.

SubSender analyzes the message. It expects a permission and a restriction to the class *InstantMessenger*. If the certificate fulfills these recommendations, the object Sender is initialized. For transmission, the certificate is encrypted with the public key K_{SND} of Sender.

The Sender object uses this permission certificate to send a message to *ForS* in order to transmit it to *Receiver*. For transmission, the certificate to *ForS* is signed with *inv(*K_{SND}*)* and encrypted afterwards with K_{FOR}, the public key of *ForS*. The kind of class is also attested using a certificate emitted by the certification authority. This certificate will be attached to the message.

ForS checks the contained permission *conc(ForS, forward)*, and whether the sender of the message identified itself as an object of class *InstantMessenger*, by comparing the declaration in the certificate to the certificate of the certification authority. If these checks are successful, the message is passed on to the *Receiver*.

6 Implementing the run-time checks using Java GuardedObjects

We now explain how this permission model can be realized in a concrete object oriented programming language such as Java.

In the JDK 1.0 security architecture, the challenges posed by mobile code were addressed by letting code from remote locations execute within a *sandbox* offering strong limitations on its execution. However, this model turned out to be too simplistic and restrictive. From JDK 1.2, a more fine-grained security architecture is employed which offers a user-definable access control, and the sophisticated concepts of signing, sealing, and guarding objects [Gon99].

A protection domain [SS75] is a set of entities accessible by a principal. In the JDK 1.2, permissions are granted to protection domains (which consist of classes and objects). Each object or class belongs to exactly one domain.

The system security policy set by the user (or a system administrator) is represented by a policy object instantiated from the class java.security.Policy. The security policy maps sets of running code (*protection domains*) to sets of access permissions given to the code. It is specified depending on the origin of the code (as given by a URL) and on the set of public keys corresponding to the private keys with which the code is signed.

There is a hierarchy of typed and parameterised access permissions, of which the root class is java.security.Permission and other permissions are subclassed either from the root class or one of its subclasses. Permissions consist of a target and an action. For file access permissions in the class FilePermission, the targets can be directories or files, and the actions include read, write, execute, and delete.

An access permission is granted if all callers in the current thread history belong to domains that have been granted the said permission. The history of a thread includes all classes on the current stack and also transitively inherits all classes in its parent thread when the current thread is created. This mechanism can be temporarily overridden using the static method doPrivileged().

Also, access modifiers protect sensitive fields of the JVM: For example, system classes cannot be replaced by subtyping since they are declared with access modifier final.

The sophisticated JDK 1.2 access control mechanisms are not so easy to use. The granting of permissions depends on the execution context (which however is overridden by doPrivileged(), which creates other subtleties). Sometimes, access control decisions rely on multiple threads. A thread may involve several protection domains. Thus it is not always easy to see if a given class will be granted a certain permission.

This complexity is increased by the new and rather powerful concepts of signed, sealed and guarded objects [Gon99]. A SignedObject contains the (to-be-)signed object and its signature.[1] It can be used internally as an authorisation token or to sign and serialise data or objects for storage outside the Java runtime. Nested SignedObjects can be used to construct sequences of signatures (similar to certificate chains).

Similarly, a SealedObject is an encrypted object ensuring confidentiality.

If the supplier of a resource is not in the same thread as the consumer, and the consumer thread cannot provide the access control context information, one can use a GuardedObject to protect access to the resource. The supplier of the resource creates an object representing the resource and a GuardedObject containing the resource object, and then hands the GuardedObject to the consumer. A specified Guard object incorporates checks that need to be met so that the resource object can be obtained. For this, the Guard interface contains the method checkGuard, taking an Object argument and performing the checks. To grant access the Guard objects simply returns, to deny access is throws a SecurityException. GuardedObjects are a quite powerful access control mechanism. However, their use can be difficult to administer [Gon99]. For example, guard objects may check the signature on a class file. This way, access to an object may be granted indirectly (and possibly unintentionally) by giving access to another object containing the signature key for which the corresponding signature provides access to the first object.

To get access to the encapsulated Object, the requesting object calls the method getObject() of the GuardedObject. In a second step, it is checked if the accessing object owns the permissions defined by the GuardObject. If it does, the method returns the reference of the encapsulated Object. The requesting object can now call any method on this object by using this reference.

The Guard normally checks the permissions by using the Java AccessController. This object reads the class of which the requesting object in an instance off the execution stack. The classes are linked to their code sources and protection domains, to which the permissions are also assigned. This means in particular that all objects of the same class own the same permissions. For permissions assigned at instantiation time this is certainly right, but if one wants to allow the delegation of permissions at run-time (as in our approach), this may lead to different sets of permissions for objects of the same class.

For this reason, it is necessary to enhance this method of permission checking. For delegating permissions dynamically, it is necessary that every object manages its certificates it received for delegation on its own. If such permissions should be considered, they must be given to the GuardedObject as a parameter when invoking the method getObject(). The Guard must thus be enhanced that it not only checks the static permissions but also the permissions contained in the certificates.

[1] Note that signing object is different from the signing of JAR files.

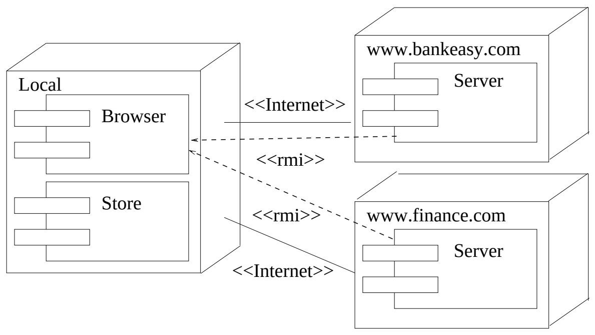

Fig. 5. Deployment diagram

It must be ensured that an object is not able to use "foreign" certificates to get access to another object. For that reason, the object references that the getObject() method produces may be secured by an asymmetric key.

If there is a permission to be delegated to a certain class, the relevant instance of the class will be referenced in the certificate. For checking that the callers' class and the named class in the certificate coincide, the callers' class will be read from the execution stack. Is there at least one certificate which is emitted for a specific object, a reference to this object must be saved in the certificate. To check the permission, the object's public key will be requested, and the reference of the encapsulated object will be encrypted with this key.

For using the reference, the caller must decode it using the corresponding private key. Since unauthorized objects do not have the appropriate private key, they are not able to decode the reference.

Another problem of the Guarded Objects in Java is that the caller gets either no or complete access to an object after the permission check. To achieve restricted access to objects, we cannot give back the real reference to an object, but build a wrapper object around the encapsulated object, having only the methods the caller has the permission for calling. These wrapper objects are the only ones which call the original object. This means that there must be created a wrapper class for all possible combinations of methods.

Note that there is one problem not to be solved by these modifications: Does one object get the reference to an encapsulated object the owner of the reference may pass it to unauthorized objects. This simply means that trusted objects must be developed in a trustworthy way.

We illustrate our approach with the example of a web-based financial application. The example was chosen to be tractable enough given the space restrictions but still realistic in that it points out some typical issues when considering access control for web-based e-commerce applications (namely to have several entities – service-providers and customers – interacting with each other while granting the other parties a limited amount of trust and by enforcing this using credentials).

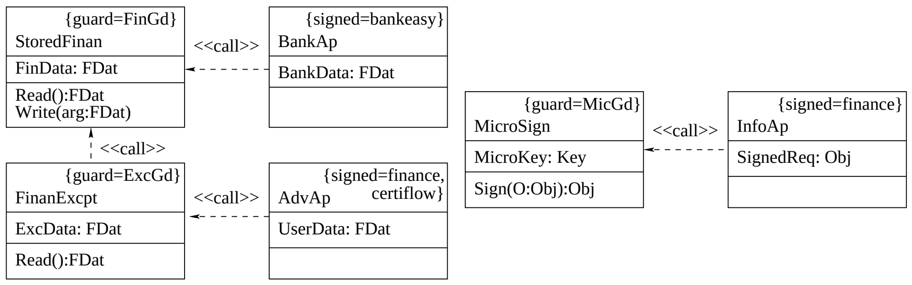

Fig. 6. Class diagram

We first describe the physical layer of the application in a UML diagram and state its security requirements. We show in UML diagrams how to employ GuardedObjects to enforce these security requirements. We prove that the specification given by the UML diagrams is secure by showing that it does not grant any access not implied by the security requirements. We end the section by giving supplementary results regarding consistency of the security requirements.

Two (fictional) institutions offer services over the Internet to local users: an Internet bank, Bankeasy, and a financial advisor, Finance. The physical layer is thus given in Fig. 5.

To make use of these services, a local client needs to grant the applets from the respective sites certain privileges.

(1) Applets that originate at and are signed by the bank can read and write the financial data stored in the local database, but only between 1 pm and 2 pm (when the user usually manages her bank account).
(2) Applets from (and signed by) the financial advisor may read an excerpt of the local financial data created for this purpose. Since this information should only be used locally, they additionally have to be signed[2] by a certification company, CertiFlow, certifying that they do not leak out information via covert channels.
(3) Applets originating at and signed by the financial advisor may use the micropayment signature key of the local user (to purchase stock rate information on behalf of the user), but this access should only be granted five times a week.

Financial data sent over the Internet is signed and sealed to ensure integrity and confidentiality. Access to the local financial data is realised using GuardedObjects. Thus the relevant part of the class diagram is given in Fig. 6.

As specified in the class diagram, the access controls are realised by the Guard objects FinGd, ExpGd and MicGd, whose behaviour is specified in Figures 7 and

[2] Here we assume that SignedObject is subclassed to allow multiple signatures on the same object [Gon99].

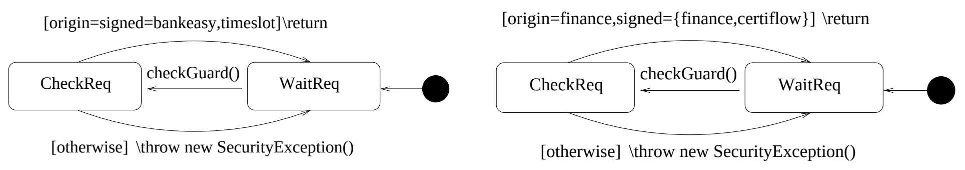
[origin=signed=bankeasy,timeslot]\return
checkGuard()
CheckReq
WaitReq
[otherwise] \throw new SecurityException()
[origin=finance,signed={finance,certiflow}] \return
checkGuard()
CheckReq
WaitReq
[otherwise] \throw new SecurityException()

Fig. 7. Statechart FinGd resp. ExcGd

8 (we assume that the condition timeslot is fulfilled if and only if the time is between 1pm and 2pm, that the condition weeklimit is fulfilled if and only if the access to the micropayment key has been granted less than five times in the current calendar week, and that the method incThisWeek increments the relevant counter).

Using the security checkers provided within the UMLsec tool suite [Too08, JY07, JS08] (Fig. 3), we can now, first, check that the specification given by UML diagrams is secure in the following sense: The specification given by UML diagrams for the guard objects does not grant any permissions not implied by the access permission requirements given in (1)–(3). Second, the tool implements the generation of the Java run-time checks in form of Guards.

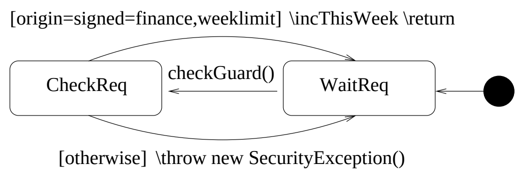
[origin=signed=finance,weeklimit] \incThisWeek \return
checkGuard()
CheckReq
WaitReq
[otherwise] \throw new SecurityException()

Fig. 8. Statechart MicGd

7 Related work

Despite a lot of work on formally verifying abstract specifications of security-critical systems, there is so far comparatively little work on making a link to the implementation level. For example, [GD04] explains how to verify a cryptographic protocol written in an abstract imperative language against security properties.

Work on formal verification of access control policies includes [SYSR06]. The difference to ours is that that paper deals with static verification, while the paper here has the goal to generate run-time checks.

Work on run-time verification in Java includes [HR04, KVK$^+$04]. Note that the current work is not run-time verification in the special sense of verification against temporal logic, but only in the wider sense of verification that is not performed at compile-time but at run-time. Instead of using temporal logic, we make use of UML statecharts to specify the properties for which run-time checks should be generated, and we make use of the GuardedObjects that are readily available in Java.

The current work is an extension of a model-based security-engineering approach [Jür04, Jür05] from the specification level to incorporate run-time checks for security permissions.

8 Conclusion

We presented an approach for application of run-time checking to enforce access control requirements at run-time in a way that enforces an overall security policy. Our approach is an improvement on the ad-hoc and error-prone way GuardedObjects are manually created and used in practice. It generates the run-time checks from a UMLsec specification of the system that captures the security policy, thereby reducing the risk for error.

References

[GD04] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Sci. Comput. Program.*, 50(1-3):73–99, 2004.

[Gon99] L. Gong. *Inside Java 2 Platform Security – Architecture, API Design, and Implementation.* Addison-Wesley, Reading, MA, 1999.

[HR04] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[JS08] J. Jürjens and J. Schreck. Automated analysis of permission-based security using UMLsec. In *FASE*, LNCS. Springer, 2008.

[Jür04] J. Jürjens. *Secure Systems Development with UML.* Springer, 2004.

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE*. IEEE, 2005.

[JY07] J. Jürjens and Yijun Yu. Tools for model-based security engineering: Models vs. code. In *22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.* ACM, 2007.

[KVK$^+$04] MoonZoo Kim, M. Viswanathan, S. Kannan, Insup Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[SS75] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SYSR06] A. Sasturkar, Ping Yang, S.D. Stoller, and C.R. Ramakrishnan. Policy analysis for administrative Role Based Access Control. In *CSFW*, pages 124–138. IEEE, 2006.

[Too08] Security verification tool, 2001-08. http://computing-research.open.ac.uk/jj/umlsectool.