STATE-DEPENDENT SECURITY DECISIONS FOR DISTRIBUTED OBJECT-SYSTEMS

Joachim Biskup
University of Dortmund
44221 Dortmund
Germany
biskup@ls6.cs.uni-dortmund.de

Thomas Leineweber*
University of Dortmund
44221 Dortmund
Germany
Thomas.Leineweber@udo.edu

Abstract  Distributed systems consist of many different objects, participating in distributed activity sequences. We present a system for state-dependent security decisions (SDSD), which can be used to specify and to enforce sets of allowed activity sequences. A specification is expressed as a protocol, which defines such a set as a regular language. The protocol is transformed into a finite automaton. Instantiated copies of the automaton are deployed by the actually participating objects, each of which has been wrapped by the SDSD-system with a security object. Knowing the automaton and based on additional control messages, the security objects can locally decide, whether a requested action is allowed or not. A running prototype implementing the SDSD-system is described, and its full integration into CORBA is outlined.

Keywords: security, distributed system, object system, state-dependent, dynamic access rights, enforcement, wrapping security objects, CORBA

*The work of Thomas Leineweber is supported by the Research Alliance Data Security North Rhine-Westphalia, http://www.datensicherheit.nrw.de/

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35587-0_24

M. S. Olivier et al. (eds.), Database and Application Security XV
© IFIP International Federation for Information Processing 2002
1. INTRODUCTION

In distributed systems there are many entities, participating in distributed activity sequences consisting of possibly many actions. In order to control the execution of these sequences of actions, special access rights are needed. These access rights could be assigned on a static basis by giving all eventually needed rights at the beginning of an activity sequence. However, this policy does not consider the fact, that the activity sequences may not be fully predetermined and that the participants may be granted more rights than needed during the whole lifespan of the sequence.

Therefore, a policy with dynamic rights is needed, which stepwise assigns the access rights only for the specific action which has to be executed next. Moreover, the set of the allowed actions at one point of time should depend on the sequence of actions executed in the past. Such dynamic rights can also be called context-aware. This context-awareness is for example found in Workflow Management Systems [Georgakopoulos et al., 1995, Vossen and Weske, 1999].

These requirements have been treated in previous theoretical work on specifying and securely enforcing allowed sequences of activities, which are to be executed in a distributed environment. Conceptually, enforcement is achieved by a finite automaton a copy of which is passed to all actually participating objects which exploit the automaton in turn with a specialized security procedure. On completion of an activity, the security procedure uses offer-, get-, put- and invoke-messages to and from other objects for initializing the next activity and for setting the automaton into the next state. Based on the given information, the objects can locally and autonomously check, which activity they are allowed to perform. A central authority is only needed to start an automaton and to resolve questions whether an object has not obeyed to the rules how to change the state of the automaton.

This paper aims at presenting and discussing a prototypical implementation of this approach. The implemented System for State-Dependent Security Decisions, SDSD-system for short, uses standard object technologies like JAVA and CORBA.

The remainder of this paper is organized as follows. In Section 2 we give a short introduction to the specification of the protocols and the automatons the SDSD-system is based on. Section 3 describes our JAVA-prototype. Security considerations are sketched in section 4. Section 5 compares our approach with other approaches for distributed rights management. And Section 6 concludes with directions for further research and development.
2. **STATE-DEPENDENT SECURITY DECISIONS (SDSD)**

The theoretical foundation of our SDSD-system has been previously presented in [Biskup and Eckert, 1994, Eckert, 1997]. In order to turn the conceptual approach into practice, we have to consider at least four kinds of phases: *object preparation phases, specification phases, instantiation phases* and *enforcement phases*.

2.1. **Object Preparation Phases**

The SDSD-system has been designed as an additional security layer to be used on top of a functional layer. The functional layer is assumed to be a distributed object system based on message passing. The functional layer may already exist before, independently of the security layer. In object-preparation-phases, the SDSD-system wraps existing or newly created objects of the underlying functional layer with additional encapsulating security objects. Whenever in the functional layer new objects are created, they have to be immediately wrapped too. Any interaction with functional objects, including interactions with a human user, are monitored by the wrapping security objects. In particular, as shown by the example given below, some of the functional objects can be understood as formal representatives of human users, and accordingly such objects act on behalf of the represented users.

2.2. **Specification Phases**

In a specification-phase, a security administrator specifies a set of allowed sequences of activities. The specification of one set of allowed sequences of

```
protocol \rightarrow \text{PROTOCOL } \text{ident;}
\quad \text{PARTICIPANTS } \text{formal}_1\text{participants;}
\quad \text{BEGIN } \text{activity } \text{END;}

\text{formal}_1\text{participants} \rightarrow \text{formal}_1\text{participant; formal}_1\text{participants}
\rightarrow \text{formal}_1\text{participant}

\text{formal}_1\text{participant activity} \rightarrow \text{ident: ident}_1\text{classname}
\rightarrow \text{step}
\rightarrow \text{activity; activity}
\rightarrow \text{activity } | \text{activity}
\rightarrow \text{activity}_1\text{activity}_2\text{activity}_3\text{activity}_4\text{activity}_5\text{activity}_6\text{activity}_7\text{activity}_8\text{activity}_9\text{activity}_10\text{activity}_11\text{activity}_12\text{activity}_13\text{activity}_14\text{activity}_15\text{activity}_16\text{activity}_17\text{activity}_18\text{activity}_19\text{activity}_20\text{activity}_21\text{activity}_22\text{activity}_23\text{activity}_24\text{activity}_25\text{activity}_26\text{activity}_27\text{activity}_28\text{activity}_29\text{activity}_30\text{activity}_31\text{activity}_32\text{activity}_33\text{activity}_34\text{activity}_35\text{activity}_36\text{activity}_37\text{activity}_38\text{activity}_39\text{activity}_40\text{activity}_41\text{activity}_42\text{activity}_43\text{activity}_44\text{activity}_45\text{activity}_46\text{activity}_47\text{activity}_48\text{activity}_49\text{activity}_50\text{activity}_51\text{activity}_52\text{activity}_53\text{activity}_54\text{activity}_55\text{activity}_56\text{activity}_57\text{activity} - \text{ident}_2\text{activator ident}_3\text{executor ident}_4\text{action}
```

*Figure 1.* Definition of the language for protocols. Words of this language describe allowed sets of activity sequences
activites is done with a so-called protocol. The definition of the language for protocols is outlined in figure 1. Each protocol, as defined by the language, has a regular expression as a core, built from triplets (activator, executor, action), named steps. Such a step specifies that an activator (object) is allowed to initiate the execution of an action at the executor (object) – but only in the contexts given by the occurrence of the step within the regular expression. This regular core of a protocol is embedded into additional features, which basically declare the formal participants in the activity sequences specified within the core.

The SDSD-system transforms a specified protocol into a finite automaton, the alphabet of which consists of the steps occurring in the protocol and which accepts exactly all specified activity sequences. An example for a protocol specification and the resulting automaton is given in figure 2.

The example-protocol has four formal participants: a Representative of the type Agent, two participants of the type Employee, namely Decider and Bookkeeper, and a fourth formal participant Data of the type ContractData. These four participants work cooperatively to establish a new insurance contract. After the Representative has inserted the contract in the Data, the Decider has to read the Data in order to decide, whether the insurance company wants to agree with the contract. If not, the contract is removed from the Data. Otherwise, the Bookkeeper waits for the payment receipt. If the contract is paid, the Data is correspondingly updated. Otherwise, the contract is removed.

2.3. Instantiation Phases

In an instantiation-phase an already specified automaton is instantiated. The instantiation consists of three parts. First, a new instance of the automaton

![Figure 2](image-url)

*Figure 2.* A specification for a protocol, defining a simplified set of activity sequences done in an insurance company while taking out a new insurance, and the resulting automaton.
is uniquely named, and second, the formal participants of the automaton are bound to some appropriate actual objects. These objects must be selected out of the collection of objects wrapped in preceding object-preparation-phases. The binding must respect all declarations, especially concerning the types. It also enforces the constraint, that different formal participants are bound to different objects. Third, a copy of the instantiated automaton (including the information of the actually bound objects) is sent to all objects which were bound. Upon reception of the copy of the instantiated automaton, the objects know, that they participate in the corresponding protocol instantiation and which other objects are participating.

2.4. Enforcement Phases

To every instantiation-phase, an enforcement-phase is assigned to. In an enforcement-phase, an actual activity sequence takes place. The objects bound to an instantiated automaton communicate with each other with offer-, get-, put- and invoke-messages for initializing any activity. In particular, as far as necessary, they get informed about the current state of the instantiated automaton. Using these messages the (wrapping security) objects can locally and autonomously check, whether the (wrapped) objects are allowed to perform a specific activity or not.

The state transition is organized as follows:

1. The actual participant that acted as the last executor sends an offer-message to all the participants, which are allowed to act as the next activator according to the instantiated automaton.

2. If a recipient of an offer-message agrees to act as activator, it replies with a get-message.

3. The last executor chooses one of the recipients that returned a get-message as the actual next activator and sends him a put-message. The put-message authorizes the chosen participant – and only him – to be the next activator within the activity sequence.

4. The activator sends an invoke-message to the associated executor. This executor is now allowed to execute the associated action. After that, the executor changes the state of the automaton to the next state according to the executed step, more precisely, the information about the new state will be encoded in the forthcoming messages.

The messages provide all information needed to locally verify, whether the instantiated automaton may change into its next state, and they are cryptographically signed. Every message includes the identification of an instantiated automaton and its current state, the identifications of the sender and the receiver,
the sequence number of the current state transition, and the signature of the last received message for this automaton. This signature is used as a link between the messages concerning the instantiated automaton, so that on demand, for instance in case of a detected security violation, a third party can reconstruct a continuous flow of messages. The offer-, get- and put-messages additionally include the identification of the planned executor and the planned action, the invoke-message includes the complete associated put-message.

The verification whether an invoke-message conforms to the instantiated automaton can be sketched as given below. Such a verification has to be performed by any security object after having received a message requiring to execute an action on the wrapped functional object. The received message should contain all information needed in addition to the instantiated automaton, in particular \( n \) gives how many state transitions have been made with this automaton up to now. \( q \) is the state before doing the actual step, \( e_l \) is the last executor, \( a_c \) is the activator and \( e_c \) is the (intended) executor in the current state transition.

1 With help of the message's signature, it is verified whether the message really originates from the actual participant given in the message.

2 The signature of the included put-message is verified.

3 The state-control checks, whether \( e_l \) which sent the corresponding offer- and put-messages was really able to change the state of the instantiated automaton to the current state. This is possible, if there exists a step in the instantiated automaton where \( e_l \) is executor and the next state of the automaton is the current state.

4 The transition control additionally checks, whether it is possible for \( e_l \) to have performed the \( n \)th activity in a sequence complying with the instantiated automaton thereby reaching the current state. This is possible, if an appropriate path can be found.

5 Now, after having recognized the current state as a (presumably)* correct one and also all other relevant parameters to be correct, the state-based access control finally checks, whether according to the instantiated automaton the present state allows the execution of the requested action.

---

* As discussed in [Biskup and Eckert, 1994, Eckert, 1997], for a full proof of correctness additional data is required. We do not treat this extension here.
3. THE PROTOTYPE

We implemented a JAVA-prototype for SDSD. The structure of an SDSD-object is shown in figure 3. It consists of a functional object, which conforms to the underlying functional layer, the wrapping security object and one automaton object per instantiated automaton the SDSD-object is participating in.

The functional object provides the actions that the SDSD-object can execute. This functional object is encapsulated in the security object. Only this security object holds a reference on the functional object, and, therefore, only it itself is able to invoke any actions on the functional object. When the security object and (possibly) the functional object (if it has not existed before) are created (object-preparation-phase), an identification, consisting of a unique name, the formal type of the functional object and (the identifiers or references for) its methods, are stored in the security object.

If an object wants to initiate a protocol (instantiation-phase) it can query such identifications. Having got this information, the object can decide on its selection which objects are to be bound to which formal participants of the protocol. Afterwards, the SDSD-system sends the instantiated automaton to the security objects of all actual participants. Each of them creates a new automaton object in order to persistently store the received instantiated automaton. Moreover, this
automaton object is responsible for the processing of those incoming messages which pertain to the stored instantiated automaton. It is also responsible for the generation of the responding messages. This can include possible human user interaction. The *automatons* block of the security object forwards incoming messages to the respective automaton objects.

The prototype performs communications among objects with the Common Object Request Broker Architecture (CORBA) [Object Management Group, b]. We use the ORB-Implementation delivered with the JDK1.3 from SUN. The initiator (object) of an activity sequence can load references to every available object, as offered by the preceding object-preparation-phases, with the use of a nameserver.

The messages between the security objects are sent using CORBA. They are implemented as CORBA valuetypes, so that they are passed to the receiver by value, not by reference, and can be queried locally without contacting the sender of the message. The methods of the security object are accessible from CORBA and handle the security decisions without interaction with the functional object but with help from the responsible automaton object.
4. SECURITY CONSIDERATIONS

Our conceptual approach requires that a functional object is not accessible without calling the wrapping security object, which shall not be bypassed. Given this property, a security proof for the conceptual approach has to demonstrate that nobody in the sequence of activities can successfully cheat without being detected.

As described in [Biskup and Eckert, 1994], there are essentially four different attacks on the automaton itself:

1. The actual executor generates a wrong successor state for the automaton.

2. The actual activator changes the state of the automaton.

3. Another SDSD-object tries to play the role of the last executor, although it is not.

4. Another SDSD-object tries to play the role of the actual activator, although it is not.

These four attacks can be detected indeed under the following additional assumptions: the signatures are unforgeable, the actions are only executed if the tests described above are successfully completed, and all participants securely log all sent and received messages. Whereas the first assumption refers to the available quality of digital signature systems, the latter assumptions basically refer to a correct implementation of the SDSD-system.

In addition to these conceptual attacks there exist further attacks on the implementation of our prototype. Apparently, the main attack tries to get a reference on a functional object, so that its methods can be called without involving the wrapping security object. Therefore, no other object than the security object shall get a reference on the functional object. This requirement is (hopefully) achieved by the claimed properties of JAVA. More generally, we have to rely on appropriate properties of the programming system used for the implementation.

Another scenario is an attempt for a denial of service attack. In this scenario, one or more malicious SDSD-objects send a lot of false messages to other SDSD-objects. This behaviour could possibly overload both the communication layer and the targeted objects. Concerning the latter situation, the targeted objects can alleviate the attack by arranging the checks such that the most easily detectable false parameters are considered first in order to generate a quick rejecting result. Such checks include for example
whether the targeted object is itself a participant in the instantiated automaton specified in the message, or

whether the sequence number of the state transition is higher than the sequence number seen when the instantiated automaton has involved this object the last time.

Further considerations about denial of service attacks lead to the implementation of additional messages used as informational messages for the SDSD-objects participating in an instantiated automaton. An end-message is sent to all participating SDSD-objects of an instantiated automaton, when the automaton reaches its halting state and therefore the activity sequence is finished. With this message the SDSD-objects know, that they can discard all messages concerning this automaton from now on. The authenticity of the end-messages is secured by checking, whether the originator of the end-message is an SDSD-object that can act as executor in the instantiated automaton and the sequence number of the last state transition is possible.

A second additional message is the revokeOffer-message. This message is sent in parallel with the put-message to all recipients of the corresponding offer-message which are not chosen as the executor. The revokeOffer-message informs the recipients, that they are not chosen and that they can ignore other messages contradicting this message. The revokeOffer-message must have the complete same data as the corresponding offer-message, otherwise it is ignored.

5. COMPARISONS

There exist many different concepts for a distributed management of access rights. An early overview of the inclusion of access control models in workflow-management systems is given in [Ellmer et al., 1994]. It gives proposals, how to integrate Discretionary or Mandatory Access Control into workflow-management systems and notes the integration of transactional systems. In [Haie et al., 2000] a system is described that is based on a simple object model with components consisting of a message handler, information repositories, a set of methods and a set of subobjects. These components reflect the notion of our security object, which knows the possible actions of the functional objects, receives and sends messages and holds type-information of the functional object. In their model the components have individual access control lists, denoting the so-called tokens needed to be allowed to invoke actions.

In [Bertino et al., 1997], a model for the assignment of both users and roles to tasks is presented, and both static and dynamic separation of duties is considered. This model is comparable to the binding of actual objects to formal participants in the SDSD-system, which also provides a separation of duties. A dynamic separation will be possible in our system with a dynamic binding of the participants. This is necessary for workflow management systems that are
able to change the workflow definition during the execution of the workflow and therefore can not bind all participants at the beginning of a workflow instance [Vossen and Weske, 1999].

[Thomas, 1997] shows with Team-based Access Control a way to distinguish between the permission assignment and a context-based permission activation. At the beginning of a task, permissions are assigned to the roles. These permissions are activated and deactivated during the execution of a workflow. A drawback is the fact, that the activation and deactivation has to be done explicitely, for example by trapping the calls on the objects and determining which permissions shall be active. Another proposal for the integration of static and dynamic access control is given in [Oh and Park, 2000]. It combines the static RBAC model [Sandhu et al., 1996] with the dynamic models given above, enabling the system to deal with workflow and non-workflow oriented tasks. It links the access not directly to the roles but uses the intermediary concept of tasks. These tasks have time constraints and and activation conditions for the access rights.

An integration of multilevel security into workflow-management systems is given in [Atluri et al., 2000]. It makes use of dependencies in the control flow between the different activities of the workflow-management system and specializes on the integration of multilevel security. It uses centralized components during the execution of the activity sequence, and it gives rights according to previous actions, based on the information on which security levels these actions were performed.

Table 1 compares the above mentioned security architectures with the SDSD-system. It is shown whether it is possible to specify allowed activity sequences, whether the rights are dependent on the previous activities, whether these rights are automatically activated and deactivated, whether a separation of duties is possible, and whether the access decision is locally done at the distributed objects without a central repository for access rights.

6. FURTHER WORK AND CONCLUSIONS

It will be necessary to lift the restriction which inhibits the system from specifying explicit parallelism for the activity sequences. This could be weakly simulated within the current language for the protocols, but only at the price of blowing up the size of the protocols and without achieving real parallelism. Therefore, an appropriate extension of the language has to be found, and ways for dealing with real parallelism in the automatons have to be investigated.

Another field for further research is to enhance the SDSD-system to deal with delayed binding of the objects to the automaton and dynamic reorganisation of the protocol, respectively the automaton. With this enhancement, the SDSD-
Table 1. Comparison of different security architectures for distributed object systems

<table>
<thead>
<tr>
<th>System</th>
<th>Set of allowed activity sequences</th>
<th>Rights dependent on previous activities</th>
<th>Automatic rights management</th>
<th>Separation of duties enforced</th>
<th>Local access decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ellmer et al., 1994</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Bertino et al., 1997</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Thomas, 1997</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Hale et al., 2000</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Oh and Park, 2000</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>possible</td>
</tr>
<tr>
<td>Atluri et al., 2000</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SDSD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

system can be integrated into workflow-management systems offering these concepts like e.g. WASA2 [Vossen and Weske, 1999].

Further considerations have to be made on the problem, how to choose the actual activator from the SDSD-objects that responded with a get-message to the offer-message. This can be done in different ways:

- A local human user chooses for the security object, which SDSD-object will be the next activator.

- Every SDSD-object gets information for the handling of the incoming get-messages. This can include a decision on a first-come-first-served-basis or on a priority basis after a given time of waiting has elapsed. These information can be given central for the object upon creation or the protocol can specify, how the SDSD-objects have to decide. These specification can even be different for different SDSD-objects.

- In the definition of the activity sequence itself the alternatives are annotated with a specification how this actual alternative shall be handled. This specification can be again on a first-come-first-serve-basis or on a priority basis.

The CORBA Security Service [Object Management Group, a] provides a framework for the integration of security services into CORBA. We want to integrate the SDSD-system into the security service. A high-level view onto a possible integration is shown in figure 5. Compared with figure 4, where the security objects are outside of the communication layer, the security objects are now embedded in the communication layer in a way such that calls to the
functional objects can not circumvent the security objects (as before with the explicit security objects in the current implementation).

![Diagram](image.png)

**Figure 5.** Possible integration into the CORBA security specification, moving the security object into the CORBA-communication layer (cf. figure 4).

These and similar extension would further enhance the achievements reached so far. The presented prototyp of the SDSD-system already allows for the dynamic, context-aware assignment of access rights, i.e. stepwise and based on previously executed actions. The running prototype is implemented in JAVA and CORBA, and thus it is portable for hopefully many applications.

**References**


