2 Background

In this chapter, we will introduce the background needed throughout this thesis. First, we will discuss some basic definitions in section 2.1. In section 2.2, we will introduce our generic notion of an Access Control (AC) model, some basic concepts, and some AC models that will be used in order to evaluate the generalizability of our approach. You may also have a look at Appendix A, presenting the terminology used in this thesis.

2.1 Information Security

This section summarizes some common security foundations and is based on Bishop *"Computer Security – Art and Science"* [25]. We will only sketch the most important concepts required later in this thesis, for a more complete discussion please refer to Bishop [25]. Please note that, for a consistent terminology, we will use terms as defined in Appendix A also for citations.

There are three principles which are considered to be the foundations for information security: confidentiality, integrity, and availability. We will follow Bishop [25] in the definition of those three concepts:

Definition 1. *Let S be a set of subjects and let R be some resource. Then R has the property of* confidentiality *with respect to S if no member of S can obtain information about R [25, p. 96].*

Informally, subjects being part of S are not allowed to see R, e. g., defining the set S as all entities not required to know R for their work. Thus, confidentiality is a principle "that information is not made available or disclosed to unauthorized individuals, entities, or processes" [1], where even knowledge over the existence of R is information about R.

Definition 2. *Let S be a set of subjects and let R be some information or a resource. Then R has the property of* integrity *with respect to S if all members of S trust R [25, p. 96].*

Informally, integrity is a principle "of safeguarding the accuracy and completeness of assets" [1]. It describes the trustworthiness subjects can

H. Petritsch, *Break-Glass,* DOI 10.1007/978-3-658-07365-7_2,

have on resources or information. It can be applied in different ways, for example, as *data integrity* for information not changed on storage or not modified during transmission, or *origin integrity*, e. g., authenticity. Mechanisms to assure integrity fall into two classes: *prevention* mechanisms detect unauthorized changes at-access and can forestall unauthorized changes in the first place, i. e., *prevent* that the system moves to a state where integrity is hurt; *detection* mechanisms may detect unauthorized changes only post-access, i. e., that a system moves to a state where integrity is hurt can be *detected* but not prevented.

Definition 3. *Let S be a set of subjects and let R be a resource. Then R has the property of* availability *with respect to S if all members of S can access R [25, p. 96].*

Thus, availability describes the ability of subjects to consume resources as planned, i. e., is the principle "of being accessible and usable upon demand by an authorized entity" [1]. Unavailability may lead to unplanned and unintended behavior. Attempts to block availability, called Denial of Service (DoS) attacks, are not only hard to prevent, but it may also be hard to detect it, e. g., to differentiate between high load and an attack.

Definition 4. *A* secure system *is a system that starts in an authorized state and cannot enter an unauthorized state [25, p. 95].*

The abstract goal is to be able to make a system secure. Security policies define what is considered to be secure.

Definition 5. *A* security policy *is a statement that partitions the states of the system into a set of* authorized*, or* secure*, states and a set of* unauthorized*, or* non-secure*, states [25, p. 95].*

Business systems manage and use confidential data and hence, have to implement confidentiality, integrity, and availability. Thus, the overall goal is to make those systems *secure*, requiring *security policies* and according mechanisms to enforce those policies, i. e., to forestall that a system can reach an unauthorized state.

The most basic questions which have to be solved are who is authorized to observe (read) what to ensure confidentiality, and who is authorized to alter (write) what to ensure integrity.

Definition 6. Authentication *is the binding of an identity to a subject [25, p. 309].*

Authentication is a technique to ensure *origin integrity*, i. e., assure that the accessing subject can be identified unambiguously. The kind of authentication being sufficient has to be defined by the security policy, e. g., requiring username and password or an authentication with cryptographic means.

Definition 7. Authorization *or* Access Control (AC) *is a technique to ensure confidentiality and a prevention mechanisms for unauthorized changes, which is a subset of data integrity.*

AC contributes to two basic security concepts: confidentiality and integrity. AC does not necessarily cover full integrity, i. e., an access may be authorized but still violate data integrity. This is why AC models are sometimes divided into confidentiality, integrity, and hybrid models. Also, AC relies on authentication as the authorization is dependent on the accessing subject. We will discuss AC in more detail in section 2.2.

Definition 8. Accountability *or* non-repudiation *is a property which ensures that a subject cannot dispute to have executed some action.*

This general property can be ensured on different levels and by different means, e. g., using cryptography to create signatures, or logging actions without letting subjects alter those logs.

The implicit goal of AC is to fulfill the *least privilege principle*, i. e., defining security policies which implement least privilege. If additional rights for specific tasks are required, those should be relinquished immediately on completion [25].

Definition 9. *The* principle of least privilege *states that a subject should be given only those privileges that it needs in order to complete its task [25, p. 343].*

Another well-known (and in most cases implicitly applied) principle is the *default deny principle* which states that all accesses not explicitly permitted are implicitly denied per default. The inverse *default permit principle*, i. e., all accesses not explicitly denied are implicitly permitted per default, is technically possible but rather exotic.

2.2 Access Control

2.2.1 Concepts

Discretionary vs. Mandatory Access Control

One very common categorization of AC model properties (or "patterns" [94], or meta models) is the differentiation between Discretionary Access Control (DAC) and Mandatory Access Control (MAC). In DAC subjects have control over resources, i. e., the discretionary to delegate permissions to other subjects. Thus, a subject "may, at his own discretion, determine who is authorized to access the objects he creates" [94]. The Trusted Computer System Evaluation Criteria (TCSEC) [81] (also known as *Orange Book*) define that DAC "shall define and control access between named users [i. e., subjects] and named objects [i. e., resource] [...]. The enforcement mechanism [...] shall allow users to specify and control sharing of those objects." As concrete example for such a DAC enforcement mechanism the concept of groups as used in file systems is mentioned. In contrast, Mandatory Access Control (MAC) (sometimes called *Non-Discretionary Access Control*) does not give this discretionary, but defines controls that are not under the control of subjects. DAC and MAC are not AC models itself, but AC models can follow one or both of them, i. e., the distinction is a conceptual one.

The AC models used by an Operating System (OS), e. g., for permissions on files, are often considered to be DAC models: a resource (e. g., a file or process) created by a subject (e. g., a user or a process) is owned by the creating subject, which has full control over the resource, e. g., can destroy it or grant other subjects access to it. Every element can be both resource and subject, e. g., a user (resource) can be created by a process (subject), or a process (resource) can be created by a user (subject). A common example for a MAC model are Multi-Level Security (MLS) systems, for which the Bell-La Padula model discussed later is one of the most prominent examples. DAC and MAC can be combined: for example, the Bell-La Padula model explicitly defines both MAC and DAC controls[1]. An example for a MAC concept implemented in operating systems is that only root can open sockets on ports below port 1024.

[1] The Bell-La Padula model is commonly referred as pure MAC model, as the MAC part is precisely modeled and the DAC part could be exchanged as long as it meets some requirements.

Delegation

Barka and Sandhu [9] characterizes delegation as a process where "some active entity in a system delegates authority to another active entity in order to carry out some functions on behalf of the former." Zhang et al. [111] identify three types of delegations: 1. backup of role, i. e., if some job function needs to be maintained by others, e. g., during absence; 2. decentralization of authortransferedity, i. e., job functions are assigned from higher to lower job positions; 3. collaboration of work, i. e., grant each other permissions to shared resources. Crampton and Khambhammettu [42] distinguish between *grant* and *transfer* delegation. For grant delegation, both the delegator (i. e., the subject possessing some permission) and the delegatee (i. e., the subject receiving some permissions) hold the delegated permission. For transfer (or *proxy* [53]) delegation, the permission is transferred and not duplicated, i. e., the delegated permission is no longer available to the delegator. Furthermore, it has to be noted that between *administrative* and *user* delegation is differentiated, e. g., Firozabadi et al. [53] differentiate between the delegation of permissions and the delegation of the right to delegate permissions.

Overall, the concept of delegation allows to modify the rights of individuals at runtime and hence allows authorized subjects to adapt required permissions of other subjects according to the current, possibly exceptional, situation. Overall, delegation is a very powerful and broad concept. For example, DAC models implicitly implement a delegation concept, as the owner of a resource can define which other users may access this resource, i. e., he may delegate some permissions to others. Consequently, there are a wide variety of concepts and models how delegation can be modeled and implemented, which will not be discussed due to space limitations.

Separation of Duty

The Separation of Duty (SoD) principle states that two actions may not be executed by the same subject. Depending on the context, there may be a scope defined where this constraint has to be valid. In the context of process models, where SoD constraints are commonly defined, the scope can be a process instance, and the actions are tasks of the process. A common example for an SoD constraint is an invoice process, where the approval has to be implemented according to the *four eyes principle*, stating that two persons have to work on an invoice to be approved. Here, for the two tasks

"acquire invoice" and "approve invoice" a SoD constraint can be defined, assuring the compliance with the four eyes principle.

AC models implementing SoD, e. g., Role Based Access Control (RBAC) as discussed in subsection 2.2.2, may differentiate between Static Separation of Duty (SSoD) and the more powerful Dynamic Separation of Duty (DSoD). For SSoD, it has to be assured that the defined privileges permit subjects to either execute the one or the other task. Thus, SSoD can be guaranteed through the assignment of permissions, and is static in the sense that it depends only on the permissions and not on the dynamic context of the application. While this does not cause overhead at runtime, it has limitations, as, e. g., there has to be a strict separation between salesmen creating invoices, and salesmen approving invoices. This is a harder constraint than the four eyes principle would require. Furthermore, in its strict interpretation the whole history of permission assignments has to be taken into account. This can cause problems when changing permissions (i. e., the assumption of static permissions does not hold), e. g., an employee moving from an executive to a controlling department. SSoD can be defined on top of, e. g., RBAC (discussed in the next subsection 2.2.2) by defining a SoD constraint between two roles. When assigning roles, it has to be assured that existing SoD constraints are not hurt.

In contrast, Dynamic Separation of Duty (DSoD) allows for more fine-grained definition of constraints, as a scope can be defined in which the constraints have to be enforced. This requires access to the system history, however, the system history may be implicitly or explicitly part of the system state (e. g., it is stored who created an invoice). Thus, to enforce DSoD at runtime, some contextual information (system history or system state) is required. For example, to ensure that the same subject cannot approve and create an invoice, it may be saved who created an invoice. The concepts of the Chinese Wall model [26] (separate resources to avoid conflict of interest) can be interpreted as implementing the SoD concept. As SoD is not in the discretion of subjects, SoD is a MAC pattern [41].

Binding of Duty (BoD) is a related concept and in some sense the opposite. It states that if one task was executed by a subject, another task has to be executed by the same subject. There is no differentiation between static and dynamic BoD, as static BoD does not make sense as it would require that permissions can only assigned to a single subject.

Obligations

We are using obligations are a concept to express conditions which have to be met but cannot be enforced by the decision making, central authority, and hence have to be enforced in a distributed way. An important distinction is the point in time when an obligation is enforced. *Pre-obligations* [39, 95] can and have to be enforced before the access. *Post-obligations* on the other hand can only be fulfilled after the access. Here, the system needs to monitor the fulfillment or satisfaction of obligations and take consequences or compensatory actions if this is not the case [23]. A related concept is the *advice*, introduced with XACML 3.0. An advice only defines what could be done, i. e., advices do not have to be enforced. This allows to model "hints" for client applications, i. e., the enforcing component may (but is not bound to) enforce the advice.

2.2.2 Access Control Models

There are a lot of AC models, both in scientific literature and commercial products. Providing a complete overview and comparison of all existing AC models would clearly exceed the frame and scope of this work. Thus, we tried to pick some models to represent some aspects of existing models. We are aware of the fact that this reflects a personal opinion. However, it is sometimes argued that all AC models can be reduced to a common basis, e. g., see [10, 35, 59] for publications dealing with this research question. We will now define a rather abstract definition of AC models.

In general, AC models describe how to map an AC request $Q \in \mathscr{Q}$ to an AC response $N \in \mathscr{N}$, i. e., $\mathscr{Q} \to \mathscr{N}$. A request Q typically contains the triple (S, R, A), with subject $S \in \mathscr{S}$, resource $R \in \mathscr{R}$, and action $A \in \mathscr{A}$. However, a request may contain fewer or further elements, depending on the concrete AC model. A response N comprises usually $\mathscr{D} \times 2^{\mathscr{O}}$, i. e., consists of a decision $D \in \mathscr{D}$ and, optionally, a, possibly empty, set of obligations $2^{\mathscr{O}} \subseteq \mathscr{O}$. We require the set $\mathscr{D}$ to contain at least the two distinctive elements PERMIT and DENY, i. e., $\{\text{PERMIT}, \text{DENY}\} \subseteq \mathscr{D} \wedge \text{PERMIT} \neq \text{DENY}$.

For the evaluation of a request Q, the

- security state $\sigma_{\text{sec}} \in \Sigma_{\text{sec}}$ captures the security relevant state of the system (i. e., the AC configuration), where σ_{sec} is an instance out of all possible security states Σ_{sec} accepted by the AC model,
- system state $\sigma_{\text{sys}} \in \Sigma_{\text{sys}}$ captures the functional state of the application or system, e. g., the formalized state of the environment, where σ_{sys} is

a concrete instance out of all possible system states Σ_{sys} accepted by the AC model,

may be required. Hence, one can define the implementation of an AC model as Access Control Function (ACF) taking a request Q, security state σ_{sec} and system state σ_{sys} as input, and returning an AC decision D with optionally a set of obligations $2^{\mathscr{O}}$, i. e.,

$$\text{ACF} : \mathscr{Q} \times \Sigma_{\text{sec}} \times \Sigma_{\text{sys}} \to \mathscr{D} \times 2^{\mathscr{O}} \tag{2.1}$$

An AC model may further define a way to define *administrative controls*. Such administrative controls (or administrative policies) allow to change the behavior of the ACF by changing the security state σ_{sec}.

Definition 10. *An* administrative control *regulates changes to the security relevant system state, i. e., defines controls for policy administration.*

Thus, we define a more general ACF_a:

$$\text{ACF}_a : \mathscr{Q} \times \Sigma_{\text{sec}} \times \Sigma_{\text{sys}} \to \mathscr{D} \times 2^{\mathscr{O}} \times \Sigma_{\text{sec}} \tag{2.2}$$

ACF_a allows to transform the security state into a new security state. For AC models which consider administrative controls as outside of the primary AC model, the ACF is sufficient. AC models using administrative controls are, e. g., models using delegation mechanisms which includes DAC models.

Access Control Matrix

The "simplest framework for describing a protection system" [25] is the *access control matrix model*, first introduced by Lampson [69] and refined by Graham and Denning [58], which we will describe here. In this model, permissions $P \subseteq \mathscr{P}$ for every subject $S \in \mathscr{S}$ on every resource $R \in \mathscr{R}$ are stored in a two-dimensional matrix M. Rows are labeled by subject names, and columns by resource names, i. e., $M[S, R]$ specifies the permissions of subject S upon resource R. Subjects are also treated as resources. The set of available permissions $\mathscr{P}$ which can be assigned depends on the applied context, common examples are *read*, *write*, *append*, *execute*, *owner* and *control*. One subject may get assigned several permissions $P \in \mathscr{P}$ for one resource, thus, the fields of the matrix contains a subset $P \subseteq \mathscr{P}$, e. g., $\{read, write\}$, or an empty set if the subject does not have any permissions on the corresponding resource.

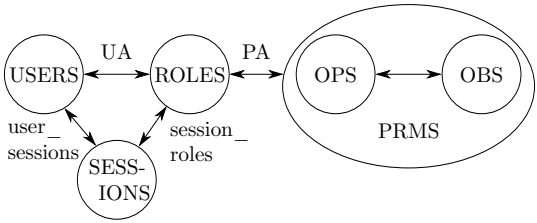

Figure 2.1: The Core RBAC model [2].

For administering the entries in M, the access control matrix follows the DAC pattern. Permissions $P \in \mathscr{P}$ can be marked with the *copy-flag*. The two permissions $\{owner, control\} \in \mathscr{P}$ have a special meaning. The copy-flag, denoted as asterisk $*$, allows a subject to copy (i. e., delegate) permissions to another subject. The permission *owner* for a resource R allows the holder of this permission to delegate and retract any permission to and from all other subject for this resource R. The permission *control* can be assigned only to fields in $M[S, R]$ where R is a subject, and allows to retract any permission from the controlled subject. The term Access Control List (ACL) refers to an approach to store the column of the matrix with the resource it protects, the term Identity Based Access Control (IBAC) can be used to characterize models managing privileges on an individual (subject) basis [63].

Role Based Access Control

Role Based Access Control (RBAC) is a well-known and established access control model, both in research and commercial systems [55]. While the concept of RBAC was already existing (e. g., Ferraiolo and Kuhn [49]), the most citet publication regarding RBAC is Sandhu et al. defining RBAC defined in [96] (RBAC96), which can be seen as the basis for the American National Standards Institute (ANSI) RBAC standard [2]. The general idea of RBAC is to group permissions (i. e., the right to execute a specific action on a specific resource) in *roles*, so that the (technical) RBAC roles reflect the (organizational) roles people can take in an organization. [2] defines four model components:

Core RBAC (see Figure 2.1) defines the fundamental data elements: subjects (USERS) are assigned to roles (ROLES), defining the user assignment (UA). Permissions (PRMS) are the approval to perform actions (operations/OPS) on resources (objects/OB), and are assigned to roles,

defining the permission assignment (PA). Sessions (SESSIONS) define an activated subset of roles for a user.

Hierarchical RBAC extends Core RBAC with a notion of a role hierarchies (RH) defining inheritance relations upon roles. If role r_1 inherits from r_2, the permissions of r_1 are a superset of the permissions of r_2, i. e., $r_1 \geq r_2$. If r_1 (r_2) is the *immediate descendant* (*ascendant*) of r_2 (r_1), one can write $r_1 >> r_2$. By convention, a diagram would show the more powerful (*senior*) role r_1 towards the top, and the less powerful (*junior*) role r_2 towards the bottom [96]. The *General Role Hierarchy* allows to define an arbitrary partial order (i. e., allowing for multi inheritance), whereas the *Limited Role Hierarchy* allows only a single immediate descendant and is therefore not supporting multiple inheritance[2].

Static Separation of Duty (SSoD) RBAC defines constraints on the user assignment (UA), i. e., SSoD prevents the assignment of roles to users which would hurt a SoD constraint. Thus, SSoD can be enforced at configuration time by administrative controls.

Dynamic Separation of Duty (DSoD) RBAC defines constraints on the activation of roles and therefore the assignment to SESSIONS, i. e., DSoD prevents the activation of a role which would hurt a SoD constraint. SSoD RBAC and DSoD RBAC can be subsumed under the term Constrained RBAC. For both SSoD and DSoD, the constraints allow to define a role set rs and a number $n, n \geq 2$, and no user may possess n roles out of rs at the same time. For example, rs could be defined as r_1, r_2, and $n = 2$. For SSoD, no user may have assigned both r_1 and r_2 in UA. In case of DSoD he may not activate both r_1 and r_2 for a session.

There are a number of further models which extend RBAC with further concepts, e. g., obligations [112], or allow to define different types of constraints, e. g., [3, 20, 21, 105]. Fuchs et al. [55] provide a survey about vast amount of RBAC-based models. For this thesis, we will limit ourself to [2] due to space reasons.

Multi-Level Security

Multi-Level Security (MLS) models follow the MAC pattern, its most prominent example is the Bell-La Padula model (although it is not a pure MLS model and therefore not a pure MAC system). The original conceptual

[2] There are some errors in the formal definition of [2] reported by, e. g., Li et al. [70]. We cited the corrected definitions.

http://www.springer.com/978-3-658-07364-0

Break-Glass
Handling Exceptional Situations in Access Control
Petritsch, H.
2014, XIII, 220 p. 15 illus., Softcover
ISBN: 978-3-658-07364-0